Supplementary material for "Computational design of nucleic acid feedback control circuits"

[†]Boyan Yordanov, [+]Jongmin Kim, [†]Rasmus L. Petersen, [‡]Angelina Shudy, [*,‡]Vishwesh V. Kulkarni, [*,†]Andrew Phillips

[†]Microsoft Research, Cambridge CB1 2FB, U.K. [+]Division of Biology and Biological Engineering, California Institute of Technology, Pasadena, CA 91125[‡]Department of Electrical Engineering, University of Minnesota, Minneapolis, MN 55455. [*]To whom correspondence should be addressed. vkulkarn@umn.edu andrew.phillips@microsoft.com

Contents

1 Proportional/Integral controllers

A *proportional/integral* (PI) controller belongs to the family of the *proportional/integral/derivative* (PID) controllers, which have been widely used in aerospace, automotive, power systems, and other fields. These controllers were first used in the 1890's to synthesize *governors* (speed limiters for engines), and their first theoretical analysis was presented by Minorsky in 1922 [6, 9]. To understand the principle behind the controllers, consider the canonical

feedback system shown in Fig. 1. The given system to be controlled is usually referred to as a *plant*, historically originating from a *process plant* or a *power plant*. Here, the objective of the controller C is to ensure that the output y of the plant P tracks the reference input r such that a desired function of the error $e = y - r$ is minimized. To achieve this, the controller C processes the error signal e and generates the controller output v, which is fed as an input to the plant. If the controller C is a PI controller, it generates the controller output v by solving the differential equation $v(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau$, where the constants K_p and K_i are tuning parameters. A number of methods to choose the values of these parameters have been developed [11, 9]. The PI controller is so named because at any time instant t, its output $v(t)$ is a weighted sum of the error signal e at that time instant and the error signal accumulated (*i.e.* integrated) over time up to that time instant. The inclusion of the integral term renders this controller *dynamic* and not *static*, which accelerates the tracking process and, in many cases, effectively eliminates the *steady state error* $\lim_{t\to\infty} e(t)$. Notably, simpler systems such as purely proportional feedback controllers cannot always meet such performance requirements.

2 Linear Time-Invariant Systems using Biomolecular Components

In the following, we outline some basic concepts regarding dynamical systems focusing specifically on linear systems and their biochemical implementations, reviewing some of the details related to the implementation strategies proposed in [8].

A dynamical biomolecular system can be viewed as a dynamical input-output system that converts the concentrations of certain chemicals into the concentrations of possibly different chemicals. A linear dynamical input-output system can be represented as

$$\dot{x} = Ax + Bu, \quad y = Cx + Du, \tag{1}$$

where $\dot{x} \doteq dx/dt$ is the time derivative of the so-called *state* $x : \mathbb{R} \to \mathbb{R}^p$ of the system, $u : \mathbb{R} \to \mathbb{R}^n$ is an input signal comprising chemical concentrations, $y : \mathbb{R} \to \mathbb{R}^m$ is an output signal comprising chemical concentrations, and $n, m, p \in \mathbb{Z}^+$ (see [2]); here, the number of inputs is n, the number of outputs is m, and the number of state variables is p. For notational simplicity, we denote a time-varying signal $u(t)$ as simply u. This system maps the given input u into the output y through the *transfer function*

$$T_{yu}(s) \doteq \frac{Y(s)}{U(s)} = C(sI - A)^{-1}B + D,$$

in the sense that $Y(s) = T_{yu}(s)U(s)$, where $Y(s) \doteq \int_{t=0}^{\infty} e^{-st} y(t) dt$ denotes the Laplace transform of the signal $y(t)$. The system described by (1) can be realized through a composition of constant gains, summation junctions, and integrators (see [2]). For example, consider an input-output system with the transfer function

$$\frac{Y(s)}{U(s)} = \frac{b_0 s + b_1}{s^2 + a_1 s + a_2}$$

This system can be described using the following set of differential equations:

$$\dot{x}_1 = x_2, \ \dot{x}_2 = -a_2 x_1 - a_1 x_2 + u, \ y = (b_2 - a_2 b_0)x_1 + (b_1 - a_1 b_0)x_2 + b_0 u.$$

As a result, this system can be realized using the block diagram shown in Fig. S1. Over the last several decades, this fact has been well used in the electrical engineering systems discipline to synthesize dynamical systems using electrical and electronic components. However, whereas an electrical signal can take negative values, a chemical concentration can take only non-negative values. Hence, we represent a chemical signal u as a pair (u^+, u^-) where u^+ and u^- are chemical species such that $u = u^+ - u^-$. To ensure a minimal representation, the following *annihilation* reaction can be enforced:

$$u^+ + u^- \xrightarrow{\mathrm{k_f}} \emptyset,$$

where $\emptyset$ is a waste product and the reaction rate k_f is arbitrarily fast.

Once a chemical representation of signals is established, various mathematical and computational operations can be implemented using chemical reactions. Our focus is specifically on molecular implementations of the summation,

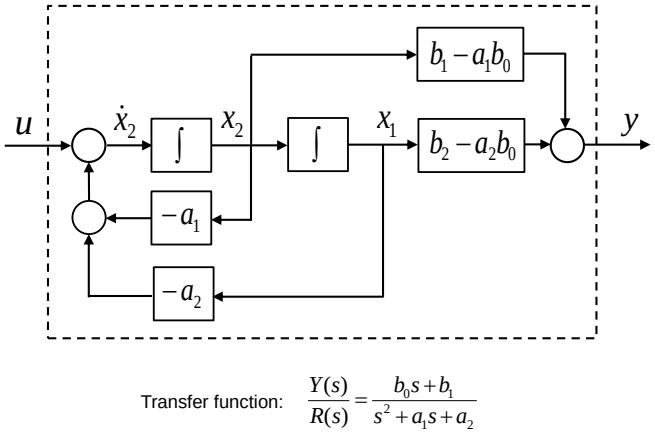


Figure S1: This block diagram illustrates how a linear dynamical system is implemented using constant gain, summation junctions, and integrators. The circle denotes an summation junction and $\int$ denotes an integrator. Oishi-Klavins have recently shown that these basic blocks can be approximated by a set of chemical reactions that is generated using 3 idealized chemical reaction types: catalysis, degradation, and annihilation.

integration, and constant gain operations, since arbitrary linear systems are decomposable into these building blocks. Recently, in [8] it was shown that the implementation of integration, gain, and summation blocks using chemical reactions can be *approximated* using the following minimal set of reaction types:

$$\begin{array}{ll} \textbf{catalysis:} & u \longrightarrow u + y \\ \textbf{degradation:} & u \longrightarrow \emptyset \\ \textbf{annihilation:} & u^+ + u^- \stackrel{\text{fast}}{\longrightarrow} \emptyset. \end{array}$$

Remark 1 For notational brevity, we shall club together reactions of the type, say, $u^+ \longrightarrow u^+ + y^+$ and $u^- \longrightarrow u^- + y^-$ and refer to those as $u^\pm \longrightarrow u^\pm + y^\pm$. □

2.1 Integrator

An integration block takes as input a signal $u(t)$ and produces the output signal $y(t) = \int_0^t u(\tau)d\tau + y(0)$, where $y(0)$ is the initial condition and $t \in \mathbb{R}$. The transfer function of an integration block is $1/s$. This block can be approximated by the following set $\mathcal{S}_I$ of idealized chemical reactions:

$$u^\pm \stackrel{\alpha}{\longrightarrow} u^\pm + y^\pm, \quad y^+ + y^- \stackrel{\eta}{\longrightarrow} \emptyset,$$

where $\alpha > 0$ and $\eta \gg \alpha$. Under this approximation, the integrator block can thus be implemented using 3 chemical reactions — two catalysis reactions and one annihilation reaction. The fact that these chemical reactions approximate the integrator can be proved as follows. The catalysis of u^+ and u^- at rate α and the annihilation of y^+ and y^- at rate η results in the following mass action equations:

$$\begin{aligned} \dot{u^+} &= \dot{u^-} = 0 \\ \dot{y^+} &= \alpha u^+ - \eta y^+ y^- \\ \dot{y^-} &= \alpha u^- - \eta y^+ y^- \\ \dot{y} &= \dot{y^+} - \dot{y^-} = \alpha u. \end{aligned}$$

Note that the bimolecular annihilation reaction drives the concentration of chemical species y^+ and y^- toward the minimal representation of the signal y, and causes the dynamics of y^+ and y^- to be nonlinear. However, the signal dynamics, $y = y^+ - y^-$, remain linear due to the symmetry between $\dot{y}^+$ and $\dot{y}^-$. As a result, by using Laplace transform, we get $sY(s) - y(0) = \alpha U(s)$, where $y(0)$ is the initial condition of the output y. As a result, it follows that the set $\mathcal{S}_I$ implements an approximation of a system that has the transfer function α/s whence an approximation of an integrator block is obtained by setting $\alpha = 1$.

2.2 Gain and Summation

A gain block transforms a given input signal u into the output signal y as $y(t) = ku(t)$ where $k \in \mathbb{R}$ is the gain. Transfer function for the gain block is k. A weighted summation block transforms a set $\{u_i\}_{i=1}^n$ of input signals into the output signal as $y(t) = \sum_{i=1}^n k_i u_i(t)$ where $k_i \in \mathbb{R}$ are the summation weights. If the summation weights are all equal to one then the weighted summation is the same as the simple summation.

Oishi-Klavins have shown in [8] that the following set $\mathcal{S}_{WS}$ of chemical reactions approximates the weighted summation block.

$$u_i^{\pm} \xrightarrow{\gamma k_i} u_i^{\pm} + y^{\pm}, \quad y^{\pm} \xrightarrow{\gamma} \emptyset, \quad y^{+} + y^{-} \xrightarrow{\eta} \emptyset,$$

where $k_i, \gamma, \eta \in \mathbb{R}^+$ for $i \in \{1, 2, \ldots, n\}$. The simple summation is obtained by setting $k_i = 1$ for all i. The gain $k > 0$ is obtained by considering only one input, i.e., by setting $i = 1$. The fact that $\mathcal{S}_{WS}$ approximates the weighted summation can be proved as follow. The idealized chemical reactions in it yield the following mass balance equations:

$$\begin{aligned}
\dot{u_i^+} &= \dot{u_i^-} = 0 \\
\dot{y^+} &= \gamma \left(\sum_{i=1}^n k_i u_i^+ - y^+ \right) - \eta y^+ y^- \\
\dot{y^-} &= \gamma \left(\sum_{i=1}^n k_i u_i^- - y^- \right) - \eta y^+ y^- \\
\dot{y} &= \gamma \left(\sum_{i=1}^n k_i u_i - y \right).
\end{aligned}$$

Taking the Laplace transforms, it follows that

$$sY(s) - y(0) = \gamma(\sum_{i=1}^n k_i U_i(s) - Y(s)).$$

Without loss of generality, assuming the initial condition $y(0)$ to be zero, we get

$$Y(s) = \frac{1}{s+\gamma} \sum_{i=1}^n k_i U_i(s).$$

Hence, using the final value theorem, we get

$$y(\infty) = \lim_{s \to 0} sY(s) = \lim_{s \to 0} \frac{s}{s+\gamma} \sum_{i=1}^n k_i U_i(s).$$

Thus, if u_i are unit step inputs then $U_i(s) = 1$ and the steady state value of the output y is $\sum_{i=1}^n k_i$.

Remark 2 The negative gain weight k_i can be obtained by replacing the catalysis reaction $u_i^{\pm} \xrightarrow{\gamma k_i} u_i^{\pm} + y^{\pm}$ with $u_i^{\pm} \xrightarrow{\gamma k_i} u_i^{\pm} + y^{\mp}$. □

Remark 3 By choosing $\gamma > 0$ arbitrarily large, the steady state can be approached arbitrarily fast and, as a result, the approximation of the weighted sum block improves by choosing $\gamma > 0$ arbitrarily large. □

3 The DSD language and simulator

Models were constructed using Visual DSD [5], a programming language for the design and analysis of DNA strand displacement devices. Visual DSD is an implementation of the programming language and compiler described in [4], and features automatic compilation of programs to strand displacement reaction networks, together with stochastic and deterministic simulation methods. Programs are written in a textual syntax, described in [4], which supports modules and local parameters to allow for abstraction and code-reuse. A DSD program defines an initial

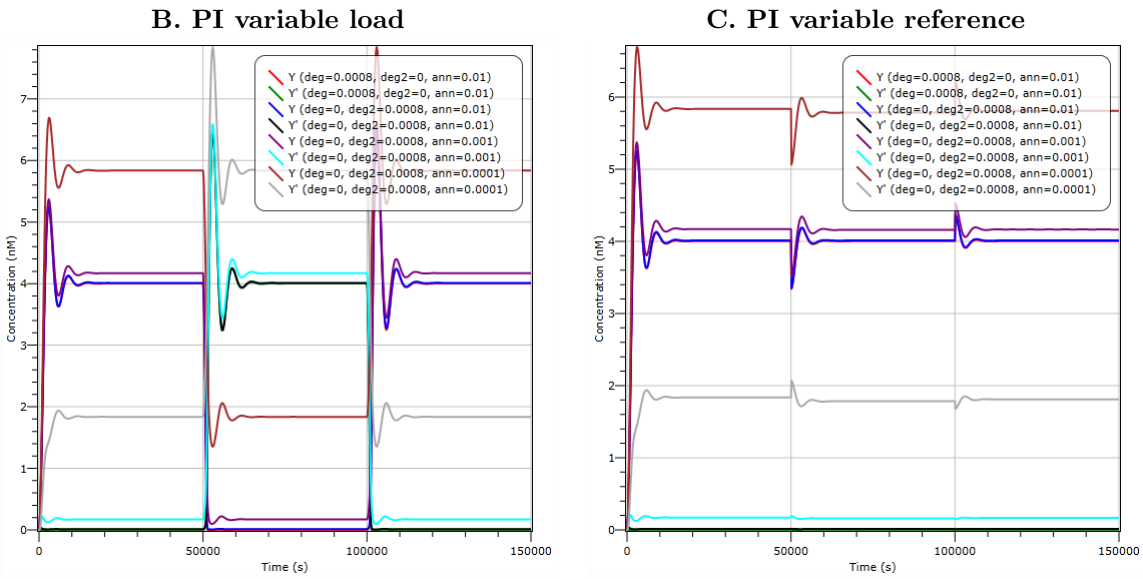


Figure S2: Signal representation through complementary species. A minimal signal representation is not realized through slow annihilation reactions between complementary species in the ideal chemical reaction controller implemented using degradation reactions, or their catalytic degradation approximations.

collection of DNA species, which can be single or double-stranded, and the DSD compiler then computes the set of all strand displacement reactions that can be generated from these initial species. The generated reactions can then be simulated using stochastic or deterministic methods. The Visual DSD language is freely available from http://research.microsoft.com/dna.

The textual syntax of the DSD programming language is defined in terms of elementary *sequences* and *species*. A sequence S comprises one or more *domains*, which can be *long domains* x or *short domains* tx^. A species can be an *upper strand* <S>, a *lower strand* {S} or a *gate* G. An upper strand <S> denotes a sequence S oriented from left to right on the page, while a lower strand {S} denotes a sequence S oriented from right to left on the page. A *double strand* [S] denotes an upper strand <S> bound to the complementary lower strand {S*}. A gate G is composed of double-stranded segments of the form {L'}<L>[S]<R>{R'}, which represents an upper strand <L S R> bound to a lower strand {L' S* R'} along the double-stranded region [S]. The overhanging sequences L, R, L' and R' can potentially be empty, in which case we simply omit them. Gates are built up by concatenating segments G1 and G2 along a common lower strand, written G1:G2. We let D range over *systems* of species. Multiple systems D1, D2 can be present in parallel, written D1|D2. We also allow module definitions of the form def X(n)=D, where n are the module parameters and X(m) is an instance of the module D with parameters n replaced by m. We assume a fixed set of module definitions, which are declared at the start of the program.

DSD Extensions We extended Visual DSD to allow the user to specify arbitrary reactions of the form $X_1 + \cdots + X_n \xrightarrow{r} X_{n+1} + \cdots + X_m$, where $X_1, \ldots, X_m$ are arbitrary DSD species, including module invocations. By default, such a reaction is assumed to have mass action kinetics with rate constant r but the rate can also be an arithmetic expression. Furthermore, if a reaction is defined inside a DSD module, then module parameters may appear within the rate expression. In principle, this allows us to express rich dynamical laws beyond mass action kinetics, for example to capture Michaelis-Menten enzyme kinetics or additional competition effects. Although these types of laws are not used explicitly in this paper, they can be written by including the relevant DSD species directly within the rate expression r. When the DSD program is compiled, hybridization and strand displacement reactions involving nucleic acids are generated from the standard DSD rules and added to the user specified reactions. Species generated by the user specified reactions are taken into account when applying the DSD rules. In this way the user specified reactions act themselves as rules, albeit very specific ones, and can thus be used to extend the behavior of a DSD model beyond the built-in strand displacement reactions, where arbitrary chemical species representing additional components such as enzymes, can be conveniently expressed. The concrete syntax for these reactions makes use of a new keyword `rxn` and has the form `rxn` n_1`*`S_1`+`$\cdots$ `+` n_k`*`S_k`->{`r`}` n_{k+1}`*`S_{k+1}`+`$\cdots$ `+` n_m`*`S_m, where $n_1, \ldots, n_m$ are integers, $S_1, \ldots, S_m$ are DSD species expressions and r is a DSD expression of type `float`. This extension also enabled the creation and editing of purely CRN models directly within Visual DSD, which could be

used for direct comparison with more complex nucleic acid designs.

The second extension was to allow the user to specify that certain toeholds are not fully matched, either by truncated toeholds or by mismatched base pairs, which is captured through a *degree of complementarity* [10]. This value is specified as a real number c (meant to be between 0 and 1) and has the effect of slowing down the binding rate by the factor c [12]. The syntax has two forms: For simple floats it is `t^`f and `t^`f`*` where f is a number of type `float` and for expressions it is `t^(`e`)` and `t^(`e`)*` where e is a DSD expression of type `float`.

The third extension was to enable the user to specify binding rates of domains consisting of adjacent toeholds. When adjacent toeholds are exposed they form a long exposed domain, the binding rate of which is not clear just from the binding rates of the individual domains. The user is provided with the syntax `dom` d `= {subdomains = [`t_1`;...;`t_n`]; bind =` r`}` which defines the composite domain d to be the concatenation of domains t_1 through t_n and have binding rate r. The composite domain d will be considered long and thus bind irreversibly with the specified rate.

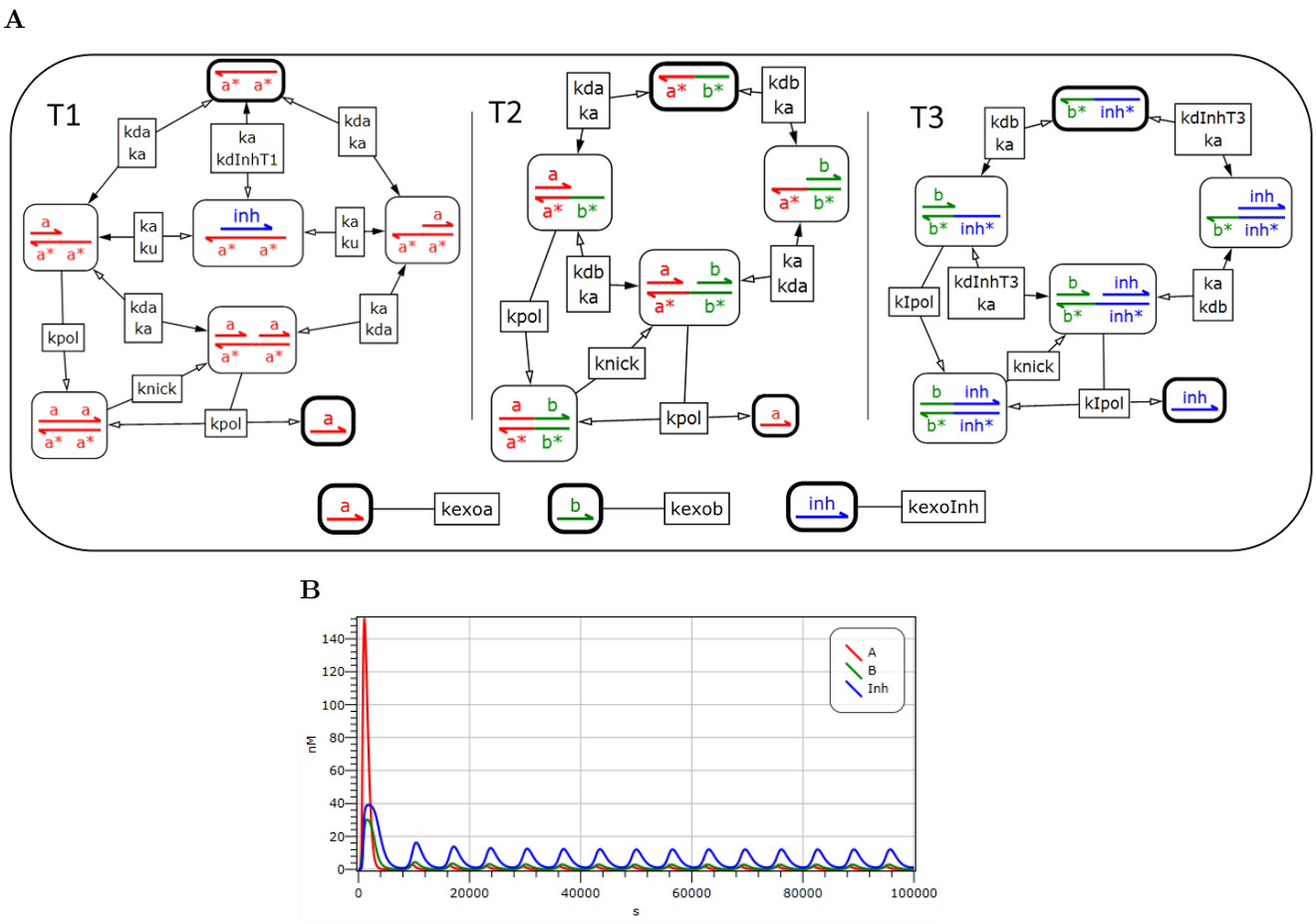


Figure S3: Visual DSD implementation of a DNA toolbox oscillator [7]. **(A)** The DNA reactions corresponding to the circuit behavior were generated and visualized automatically from the Visual DSD code. The graphical representation was further adjusted to improve readability. **(B)** Simulations of the model reveal the expected oscillatory behavior.

4 Visual DSD implementation of a DNA toolbox oscillator

We implemented a previously published DNA toolbox oscillator [7] in Visual DSD (Fig. S3). Kinetic parameters were obtained from the published model [7], and all reactions were generated automatically by Visual DSD and shown to be consistent with the manually constructed model [7]. The Visual DSD code for this system is reproduced below.

```
directive parameters
  [ ka = 4.3333e-04
  ; kda = 0.0383
  ; kdb = 0.0135
  ; kdInhT1 = 9.5e-5
  ; kdInhT3 = 3.5e-5
  ; kpol = 0.2833
  ; kIpol = 0.1150
  ; knick = 0.05
  ; kexoa = 0.0053
  ; kexob = 0.0062
  ; kexoInh = 0.02
  ; ku = 0.0
  ]
directive unproductive
directive simulation deterministicstiff
```

```
directive duration 100000.0 points 1000
directive plot  A(); B(); Inh()

dom a = {bind = ka; unbind = kda; colour = "red"}
dom b = {bind = ka; unbind = kdb; colour = "green"}
dom inh = {bind = ka; unbind = kdInhT3; colour = "blue"}

def Signal(x) = <x^>
def Primed(x,y) = [x^]{y^*}
def Extended(x,y) = [x^ y^]
def Template(x,y) = {x^* y^*}
def LHTemplate(x,y) = [x^]{y^*}
def RHTemplate(x,y) = {x^*}[y^]
def Nicked(x,y) = [x^]:[y^]
def Inhibited(x,y) = [y^ x y^]

def Activation(N,x,y,kp,kn) =
( N * Template(x,y)
| rxn Primed(x,y) ->{kp} Extended(x,y)
| rxn Extended(x,y) ->{kn} Nicked(x,y)
| rxn Nicked(x,y) ->{kp} Extended(x,y) + Signal(y)
| 0*Primed(x,y)
| 0*Extended(x,y)
| 0*Nicked(x,y)
)
def Repression(N,x,y,kp,kn) =
( Activation(N,y,y,kp,kn)
| rxn Signal(x) + Template(y,y) <->{ka,kdInhT1} Inhibited(x,y)
| rxn Signal(x) + LHTemplate(y,y) <->{ka,ku} Inhibited(x,y) + Signal(y)
| rxn Signal(x) + RHTemplate(y,y) <->{ka,ku} Inhibited(x,y) + Signal(y)
| 0*Inhibited(x,y)
)
def Exo(x,k) =(rxn Signal(x) ->{k} )
def A() = Signal(a)
def B() = Signal(b)
def Inh() = Signal(inh)
def Oscillator() =
( 0.1 * A()
| 0.1 * B()
| 0.1 * Inh()
| Activation(5.0,a,b,kpol,knick)
| Activation(30.0,b,inh,kIpol,knick)
| Repression (30.0,inh,a,kpol,knick)
| Exo(a,kexoa)
| Exo(b,kexob)
| Exo(inh,kexoInh)
)
def Catalysis() =
( 0.1 * A()
| 0.1 * B()
| Activation(5.0,a,b,kpol,knick)
)

Oscillator()
(* Catalysis() *)
```

5 Visual DSD implementation of a genelet oscillator

We implemented a previously published genelet oscillator [3] in Visual DSD (Fig. S4). Kinetic parameters were obtained from the published model, and all reactions generated automatically by Visual DSD were shown to be consistent with those of the previously manually constructed model [3]. The Visual DSD code for this system is shown below.

```
directive compilation infinite
directive concentration nM
directive polymers
directive parameters [
kTA21 = 74000.0e-9;
kTA12 = 14000.0e-9;
kAI1 = 53000.0e-9;
krAI1 = 24000.0e-9;
kAI2 = 31000.0e-9;
kTAI21 = 28000.0e-9;
kTAI12 = 140000.0e-9;
kAIrA1 = 28000.0e-9;
cRNaseH = 1.5e+1;
cRNAP = 1.25e+2;
kRNAP = 0.0323;
kRNaseH = 0.0196;
S = 0.02
]

(*directive sweep S = [0.1,0.09,0.08,0.07,0.06,0.05,0.04,0.02,0.01]*)
directive sample 72000.0 1000
directive simulation deterministicstiff
directive plot T21(); T12() (*; rI2(); A2() *)
dom ta2 = {bind = kTAI12; colour = "lightblue"}
dom a2 = {bind = kTA12; colour = "red"}
dom t = {bind = 0.0; colour = "blue"} (* negligible binding rate *)
dom p = {bind = 0.0; colour = "purple"} (* negligible binding rate *)
dom ta1 = {bind = kTAI21; colour = "darkgreen"}
dom a1 = {bind = kTA21; colour = "magenta" }
dom tdI1 = {bind = kAIrA1; colour = "black"}
dom AI1 = {bind = kAI1; subdomains = [p;a1;ta1]}
dom AI2 = {bind = kAI2; subdomains = [ta2;a2;t]}
dom rAI1 = {bind = krAI1; subdomains = [a1;ta1;tdI1]}
dom TA12 = {bind = kTA12; subdomains = [a2;t] }
dom TA21 = {bind = kTA21; subdomains = [t*;p;a1] }
(*
dom prom = {colour = "blue"}
dom h = {colour = "brown"}
dom trI2 = {colour = "lightgreen"}
*)

new RNaseH new RNaseHA2rI2 new RNaseHrA1dI1
new RNAP new RNAPT12A2 new RNAPT21A1 new RNAPT12 new RNAPT21
def RNaseH() = <RNaseH>
def RNaseHA2rI2() = <RNaseHA2rI2>
def RNaseHrA1dI1() = <RNaseHrA1dI1>
def RNAP() = <RNAP>
def RNAPT12A2() = <RNAPT12A2>
def RNAPT21A1() = <RNAPT21A1>
def RNAPT12() = <RNAPT12>
def RNAPT21() = <RNAPT21>

def rA1() = <tra1 a1^ ta1^ tdI1^ t^ h>
def A1() =  <t^* p^ a1^ ta1^>
def dI1() =   <tdI1^* ta1^* a1^* p^*>
def A2() = <t^* a2^* ta2^*>
```

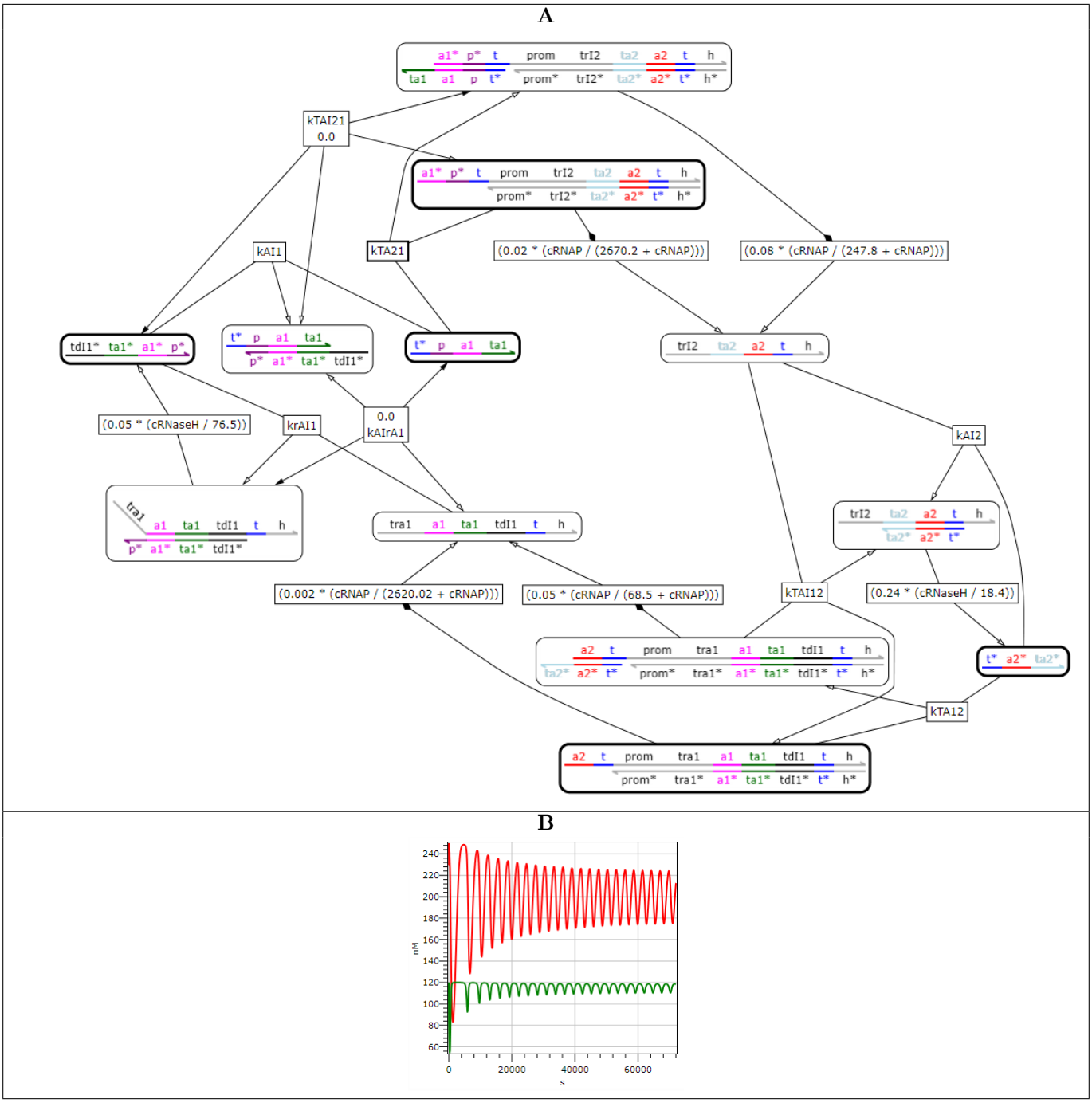


Figure S4: Visual DSD implementation of a genelet oscillator [3]. **(A)** The DNA reactions corresponding to the circuit behavior were generated and visualized automatically from the Visual DSD code. The layout of the graphical representation was further adjusted to improve readability. **(B)** Simulations of the model reproduced the previously obtained oscillatory behavior.

```
def rI2() = <trI2 ta2^ a2^ t^ h>
def A2rI2() = <trI2>[ta2^ a2^ t^]<h>
def rA1dI1() = <tra1>{p^*}[a1^ ta1^ tdI1^]<t^ h>
def A1dI1() = <t^*>[p^ a1^ ta1^]{tdI1^*}
def T12() = <a2^ t^>[prom tra1 a1^ ta1^ tdI1^ t^ h]
def T12A2() = {ta2^*}[a2^ t^]::[prom tra1 a1^ ta1^ tdI1^ t^ h]
def T21() = <a1^* p^* t^>[prom trI2 ta2^ a2^ t^ h]
def T21A1() = {ta1^}[a1^* p^* t^]::[prom trI2 ta2^ a2^ t^ h]
def T11A2() = {ta2^*}[a2^ t^]::[prom trI2 ta2^ a2^ t^ h]
def T11() = <a2^ t^>[prom trI2 ta2^ a2^ t^ h]


def kplus = 1.0e-1
def kplusH = 1.0e-1
def kminusON12 = 6.8
def kminusOFF12 = 262.0
def kminusON21 = 24.7
def kminusOFF21 = 267.0
def kminusH1 = 7.6
def kminusH2 = 1.6
def kcatON12 = 0.05
def kcatOFF12 = 0.002
def kcatON21 = 0.08
def kcatOFF21 = 0.02
def kcatH1 = 0.05
def kcatH2 = 0.24

def KMH1 = (kminusH1 + kcatH1) / kplusH
def KMH2 = (kminusH2 + kcatH2) / kplusH
def KMON12 = (kminusON12 + kcatON12) / kplus
def KMON21 = (kminusON21 + kcatON21) / kplus
def KMOFF12 = (kminusOFF12 + kcatOFF12) / kplus
def KMOFF21 = (kminusOFF21 + kcatOFF21) / kplus

def Activation() =
( 1.2e+2*T11()
| 5.0e+2*A2()
| 0*A2rI2()
| 0*T11A2()
| rxn A2rI2() ->{kRNaseH} A2()
| rxn T11A2() ->{kRNAP} T11A2() + rI2()
| 0*rI2()
)
def RNaseH_enzymatic() =
( 1.5e+1*RNaseH()
| rxn RNaseH() + A2rI2() <->{kplusH}{kminusH2} RNaseHA2rI2()
| rxn RNaseHA2rI2() ->{kcatH2} RNaseH() + A2()
| rxn RNaseH() + rA1dI1() <->{kplusH}{kminusH1} RNaseHrA1dI1()
| rxn RNaseHrA1dI1() ->{kcatH1} RNaseH() + dI1()
)
def RNaseH_firstorder() =
( rxn A2rI2() ->{S*kcatH2*cRNaseH/KMH2} A2()
| rxn rA1dI1() ->{S*kcatH1*cRNaseH/KMH1} dI1()
)
def RNAP_enzymatic() =
( 1.25e+2*RNAP()
| rxn RNAP() + T12A2() <->{kplus}{kminusON12} RNAPT12A2()
| rxn RNAPT12A2() ->{kcatON12} RNAP() + T12A2() + rA1()
| rxn RNAP() + T21A1() <->{kplus}{kminusON21} RNAPT21A1()
| rxn RNAPT21A1() ->{kcatON21} RNAP() + T21A1() + rI2()
| rxn RNAP() + T12() <->{kplus}{kminusOFF12} RNAPT12()
| rxn RNAPT12() ->{kcatOFF12} RNAP() + T12() + rA1()
```

```
| rxn RNAP() + T21() <->{kplus}{kminusOFF21} RNAPT21()
| rxn RNAPT21() ->{kcatOFF21} RNAP() + T21() + rI2()
)
def RNAP_firstorder() =
( rxn T12A2() ->{kcatON12*cRNAP/(KMON12 + cRNAP)} T12A2() + rA1()
| rxn T21A1() ->{kcatON21*cRNAP/(KMON21 + cRNAP)} T21A1() + rI2()
| rxn T12() ->{kcatOFF12*cRNAP/(KMOFF12 + cRNAP)} T12() + rA1()
| rxn T21() ->{kcatOFF21*cRNAP/(KMOFF21 + cRNAP)} T21() + rI2()
)
def System() =
( 1.2e+2*T12()
| 5.0e+2*A2()
| 2.5e+2*T21()
| 2.5e+2*A1()
| 1.0e+3*dI1()
| 0*rA1()
| 0*rI2()
| 0*A2rI2()
| 0*T12A2()
| 0*T21A1()
| 0*rA1dI1()
| 0*A1dI1()
| RNaseH_firstorder()
(* | RNaseH_enzymatic() *)
| RNAP_firstorder()
(* | RNAP_enzymatic() *)
)

System()
(* Activation() *)
```

6 Visual DSD code for PI controller implementations

In the following, we summarize the four-domain implementation of the PI controller, as well as the two-domain DNA strand displacement, DNA enzyme and RNA enzyme implementations. In order to enable a consistent comparison across the three designs, the plant is modeled using the same set of idealized chemical reactions.

6.1 Chemical Reaction Network implementation

```
(*------------------ Settings ----------------------------------*)
directive declare
directive simulation deterministicstiff
directive parameters
  [ deg = 0.0008
  ; cat = 0.0008
  ; pol = 0.0167
  ; nick = 0.0167
  ; x0 = 8.0 (* 4.0, 8.0 *)
  ; ann = 0.01 (* 0.01, 0.001 *)
  ; bind = 5.4e-6
  ; bind2 = 0.001  (* 0.001, 5.4e-6 *)
  ; bind1 = 0.00005
  ; unbind = 0.1126
  ; Cmax = 1000.0
  ; c = 0.0008
  ; kt = 0.001
  ; ku = 0.001
  ; s = 2.0
  ]
(* Circuit component settings *)
(*
directive duration 10000.0 points 1000
directive plot X(); X'(); Y(); Y'(); R();
E(); E'(); V(); V'()
*)
(* PI Controller settings *)

directive duration 250000.0 points 1000
directive plot
  sub (R(); R'());
  sub (V(); V'());
  sub (Y(); Y'());
  sub (Load(); Load'())

directive event R'() 2.0*x0 @  50000.0
directive event R() 2.0*x0 @ 100000.0
directive event <load> 2.0 @ 150000.0
directive event <load'> 1.0 @ 200000.0
(*
directive duration 500000.0 points 1000
directive event Signal'(r)   2*x0 @  50000.0
directive event Signal(r)    2*x0 @ 100000.0
directive event Signal'(r)   2*x0 @ 150000.0
directive event Signal(r)     2*x0 @ 200000.0
directive event Signal'(r)   2*x0 @ 250000.0
directive event Signal(r)     2*x0 @ 300000.0
directive event Signal'(r)  2*x0 @ 350000.0
directive event Signal(r)     2*x0 @ 400000.0
directive event Signal'(r)  2*x0 @ 450000.0
*)

(*------------------ CRN ------------------*)
new x new x' new v new v' new y new y' new r new r'
```

```
new e new e' new load new load'
def Signal((x,x')) = <x>
def Signal'((x,x')) = Signal((x',x))
def Degradation(k,x) = rxn Signal(x) ->{deg/k}
def Degradation'(k,(x,x')) = Degradation(k,(x',x))
def Catalysis(k,x,y) = rxn Signal(x) ->{cat*k} Signal(x) + Signal(y)
def Catalysis'(k,(x,x'),(y,y')) = Catalysis(k,(x',x),(y',y))
def CatalysisInv(k,x,(y,y')) = Catalysis(k,x,(y',y))
def CatalysisInv'(k,(x,x'),y) = Catalysis(k,(x',x),y)
def Annihilation(x) =  rxn Signal(x) + Signal'(x) ->{ann}
def x = (x,x')
def y = (y,y')
def v = (v,v')
def e = (e,e')
def r = (r,r')
(*------------------ Blocks --------------------------------*)
def Catalysis2(k,x,y) =
( Catalysis(k,x,y)
| Catalysis'(k,x,y)
)
def Catalysis2Inv(k,x,y) =
( CatalysisInv(k,x,y)
| CatalysisInv'(k,x,y)
)
def Degradation2(k,x) =
( Degradation(k,x)
| Degradation'(k,x)
)
def Integration(kI,x,y) = Catalysis2(kI,x,y)
def Gain(k,kD,x,y) =
( Catalysis2(k,x,y)
| Degradation2(kD,y)
)
def Summation(k1,k2,kD,x1,x2,y) =
( Catalysis2(k1,x1,y)
| Catalysis2(k2,x2,y)
| Degradation2(kD,y)
)
def SummationInv(k1,k2,kD,x1,x2,y) =
( Catalysis2(k1,x1,y)
| Catalysis2Inv(k2,x2,y)
| Degradation2(kD,y)
)
def PI(kP,kI, r, y, v,e) =
( SummationInv(1.0,1.0,1.0,r,y,e)
| Integration(kI,e,x)
| Summation(kP,1.0,1.0,e,x,v)
| Annihilation(x)
)
def R()  = Signal(r)
def R'() = Signal'(r)
def E()  = Signal(e)
def E'() = Signal'(e)
def V()  = Signal(v)
def V'() = Signal'(v)
def Y()  = Signal(y)
def Y'() = Signal'(y)
def X() = Signal(x)
def X'() = Signal'(x)
def Load() = <load>
def Load'() = <load'>
def Plant(v, y) =
```

```
( rxn Signal(v)  ->{0.2} Signal(v)  + Signal(y)
| rxn Signal'(v) ->{0.2} Signal'(v) + Signal'(y)
| rxn Signal(y)  ->{0.1}
| rxn Signal'(y) ->{0.1}
| rxn Signal'(y) + Signal(y) ->{0.1}
| rxn Load() + Signal(y)   ->{0.01} Load()
| rxn Load'() + Signal'(y) ->{0.01} Load'()
| rxn Load() + Load'() ->
)
(*------------------ Full system  ----------------------------*)
def System() =
( 0 * V() | 0 * V'()
| 0 * Y() | 0 * Y'()
| 0 * Load() | 0 * Load'()
| x0* R() | 0 * R'()
| PI(1.0,1.0,r,y,v,e)
| Plant(v,y)
| Annihilation(r)
| Annihilation(v)
| Annihilation(e)
)

(*------------------ Tests ----------------------------------*)
def TestAnnihilation() =
( 2 * X() | X'()
| Annihilation(x)
)
def TestCatalysis() =
( X() | 0 * Y()
| Catalysis(1.0,x,y)
)
def TestDegradation() =
( X() | 0 * X'()
| Degradation(1.0,x)
(*| Annihilation(x) *)
)
def TestGain() =
( X() | 0 * Y() | 0 * Y'()
| Gain(1.0,2.0,x,y)
| Annihilation(y)
)
def TestSummation() =
( 2 * X() | E() | 0 * V() | 0 * V'()
| Summation(1.0,1.0,1.0,x,e,v)
| Annihilation(v)
)
def TestSummationInv() =
( 2 * Y() | R() | 0 * E() | 0 * E'()
| SummationInv(1.0,1.0,1.0,y,r,e)
| Annihilation(e)
)
(*------------------ Run ------------------------------------*)
System()
(*
TestAnnihilation()
TestCatalysis()
TestDegradation()
TestGain()
TestSummation()
TestSummationInv()
*)
```

6.2 4Domain DNA Strand Displacement implementation

The Visual DSD code for this system is identical to that of the CRN code, but with the CRN section replaced with the following, where kinetic parameters for this system were obtained from [8] and were consistent with published experimental results [12, 1]

```
directive compilation infinite

dom x1 = {colour = "red"; bind = kt} dom x1' = {colour = "red"; bind = kt}
dom x3 = {colour = "red"; bind = kt} dom x3' = {colour = "red"; bind = kt}
dom r1 = {colour = "purple"; bind = kt} dom r1' = {colour = "purple"; bind = kt}
dom r3 = {colour = "purple"; bind = kt} dom r3' = {colour = "purple"; bind = kt}
dom e1 = {colour = "blue"; bind = kt} dom e1' = {colour = "blue"; bind = kt}
dom e3 = {colour = "blue"; bind = kt} dom e3' = {colour = "blue"; bind = kt}
dom v1 = {colour = "orange"; bind = kt} dom v1' = {colour = "orange"; bind = kt}
dom v3 = {colour = "orange"; bind = kt} dom v3' = {colour = "orange"; bind = kt}
dom y1 = {colour = "green"; bind = kt} dom y1' = {colour = "green"; bind = kt}
dom y3 = {colour = "green"; bind = kt} dom y3' = {colour = "green"; bind = kt}
dom load = {colour = "black"; bind = kt} dom load' = {colour = "black"; bind = kt}
dom load = {colour = "black"; bind = kt} dom load' = {colour = "black"; bind = kt}
dom t = {colour = "magenta"; bind = kt; unbind = unbind}

new xh new x2 new xh' new x2' new vh new v2 new vh' new v2' new yh new y2 new yh' new y2'
new rh new r2 new rh' new r2' new eh new e2 new eh' new e2'

def Signal0((xh,x1,x2,x3),x') = <xh x1^ x2 x3^>
def Degradation0(k,(xh,x1,x2,x3),x') = Cmax/k * {x1^(c)*}[x2 x3^]
def Annihilation0((xh,x1,x2,x3),(xh',x1',x2',x3')) =
( Cmax * {x1^*}[x2 x3^ x1'^]:[x2' x3'^]
| Cmax * <x2 x3^ x1'^>
| Cmax * {x1'^*}[x2' x3'^ x1^]:[x2 x3^]
| Cmax * <x2' x3'^ x1^>
)
def Catalysis0(k,(xh,x1,x2,x3),x',(yh,y1,y2,y3),y') =
( Cmax*k * {x1^(c)*}[x2 x3^]<yh y1^ xh x1^>
| Cmax*k * {x3^*}[yh y1^]<y2 y3^>:[xh x1^]<x2 x3^>
)
def Signal((x,x')) = Signal0(x,x')
def Degradation(k,(x,x')) = Degradation0(k,x,x')
def Catalysis(k,(x,x'),(y,y')) = Catalysis0(k,x,x',y,y')
def Annihilation((x,x')) = Annihilation0(x,x')
def Signal'((x,x')) = Signal((x',x))
def Degradation'(k,(x,x')) = Degradation(k,(x',x))
def Catalysis'(k,(x,x'),(y,y')) = Catalysis(k,(x',x),(y',y))
def CatalysisInv(k,x,(y,y')) = Catalysis(k,x,(y',y))
def CatalysisInv'(k,(x,x'),y) = Catalysis(k,(x',x),y)

def x = ((xh,x1,x2,x3),(xh',x1',x2',x3'))
def y = ((yh,y1,y2,y3),(yh',y1',y2',y3'))
def v = ((vh,v1,v2,v3),(vh',v1',v2',v3'))
def e = ((eh,e1,e2,e3),(eh',e1',e2',e3'))
def r = ((rh,r1,r2,r3),(rh',r1',r2',r3'))
```

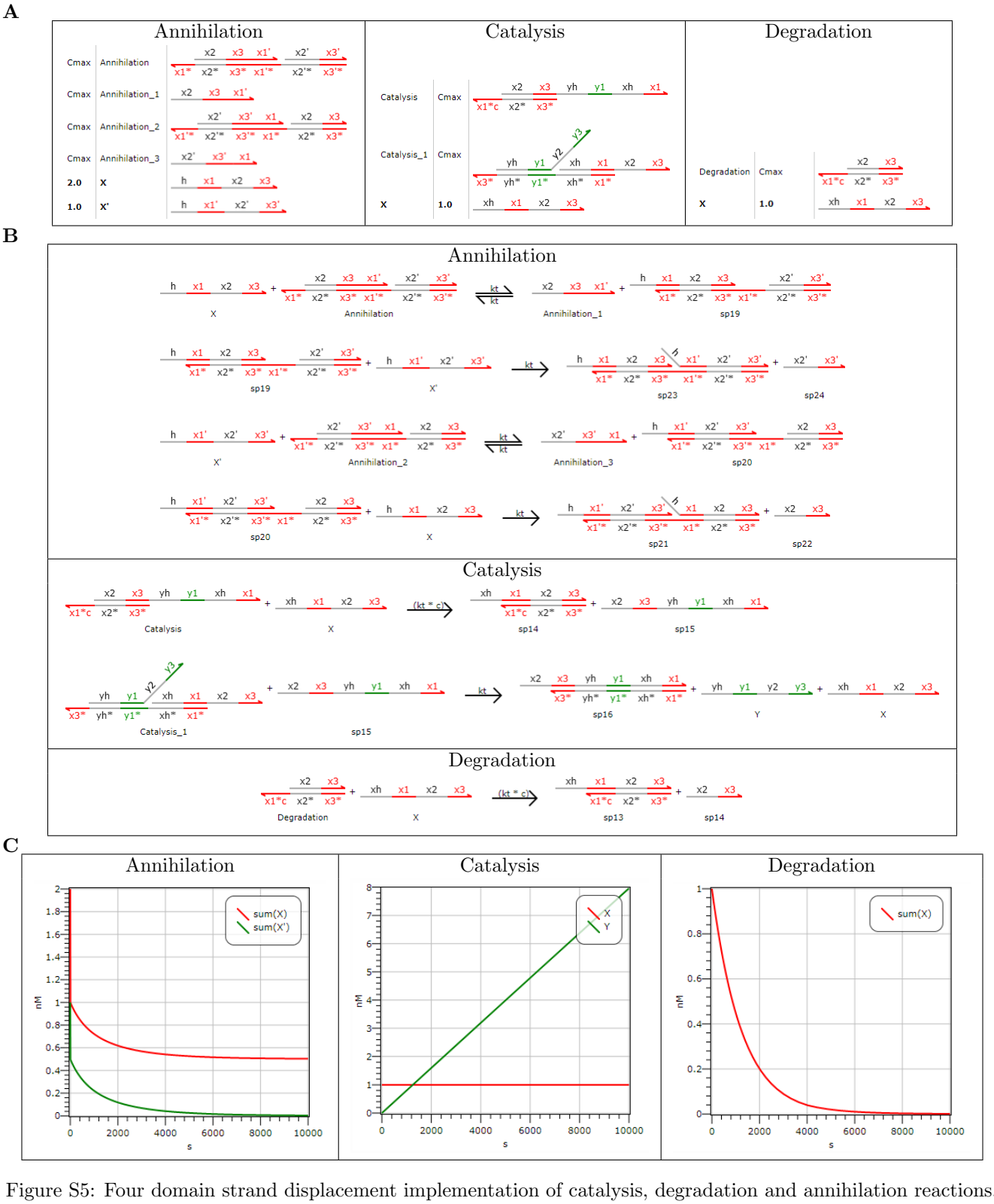


Figure S5: Four domain strand displacement implementation of catalysis, degradation and annihilation reactions.

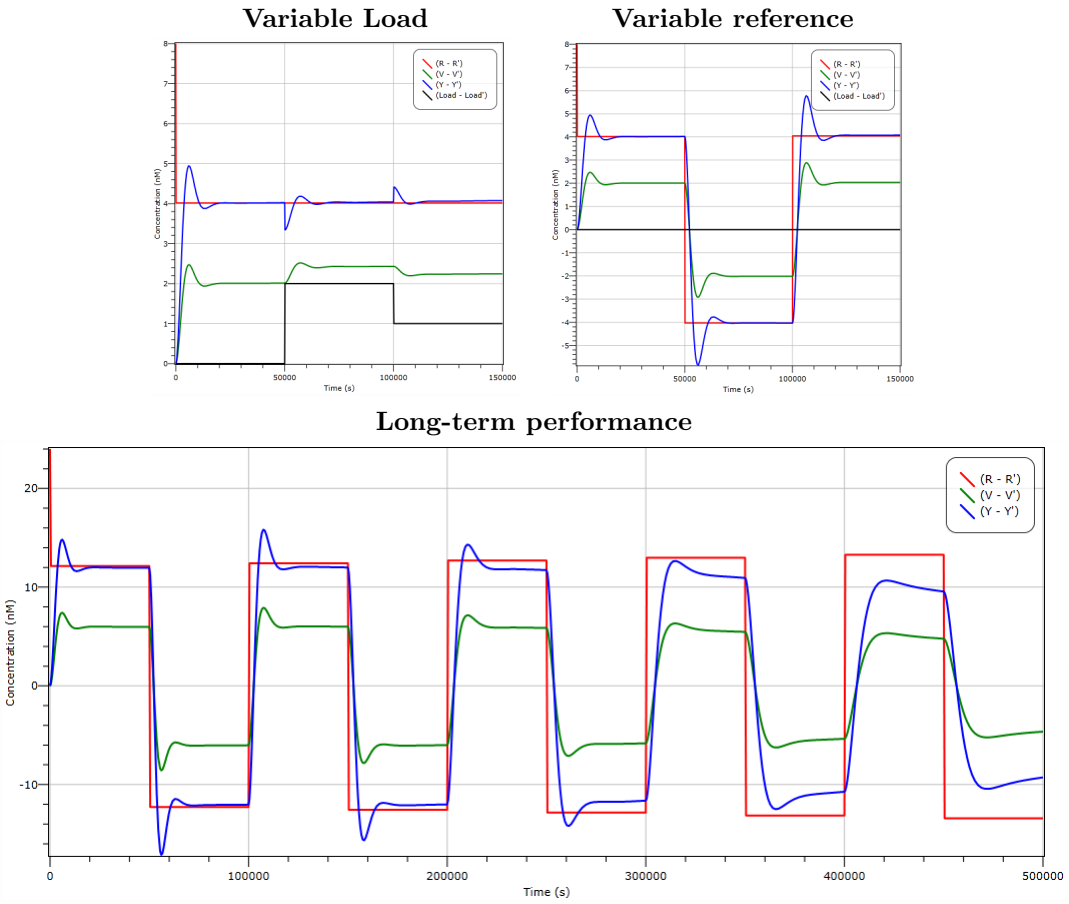
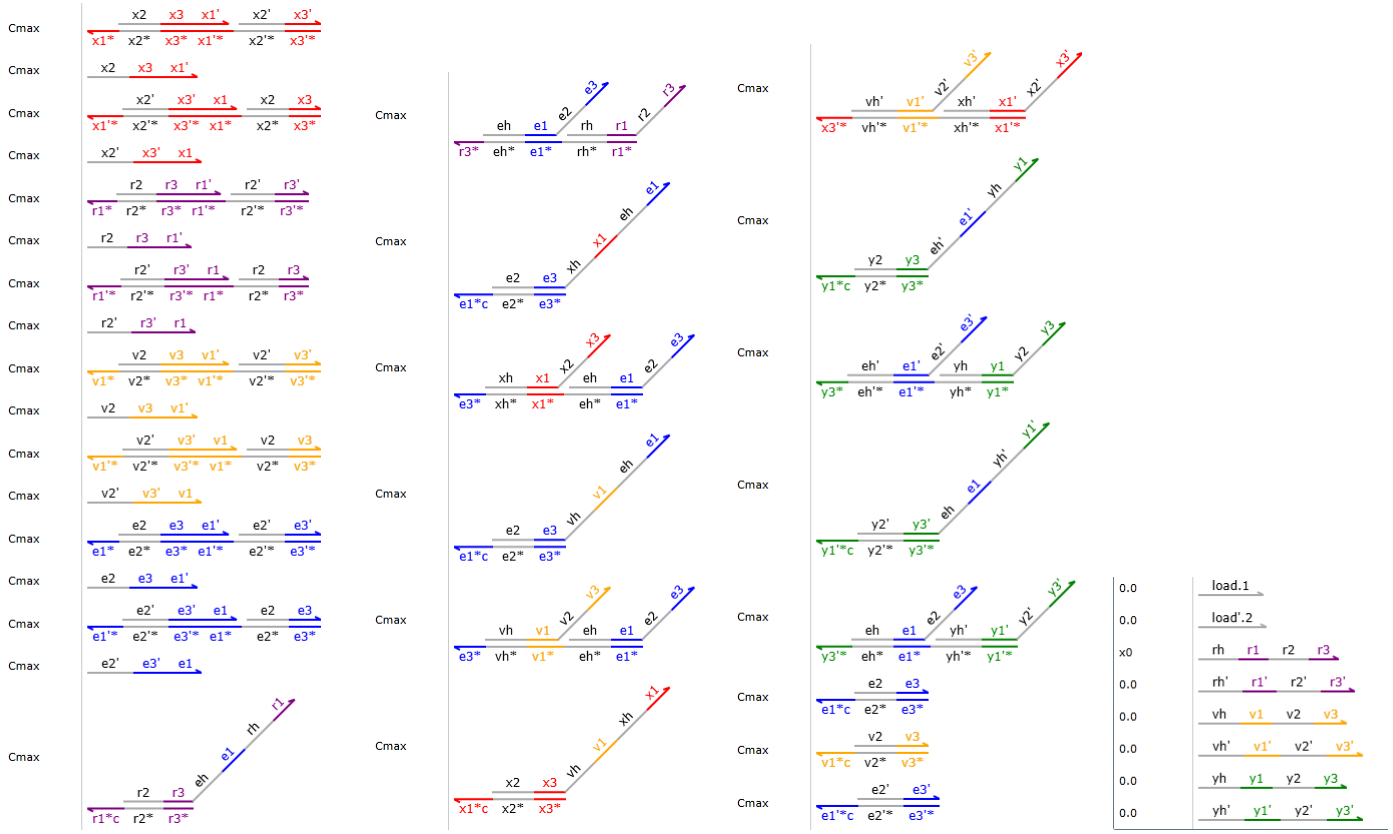
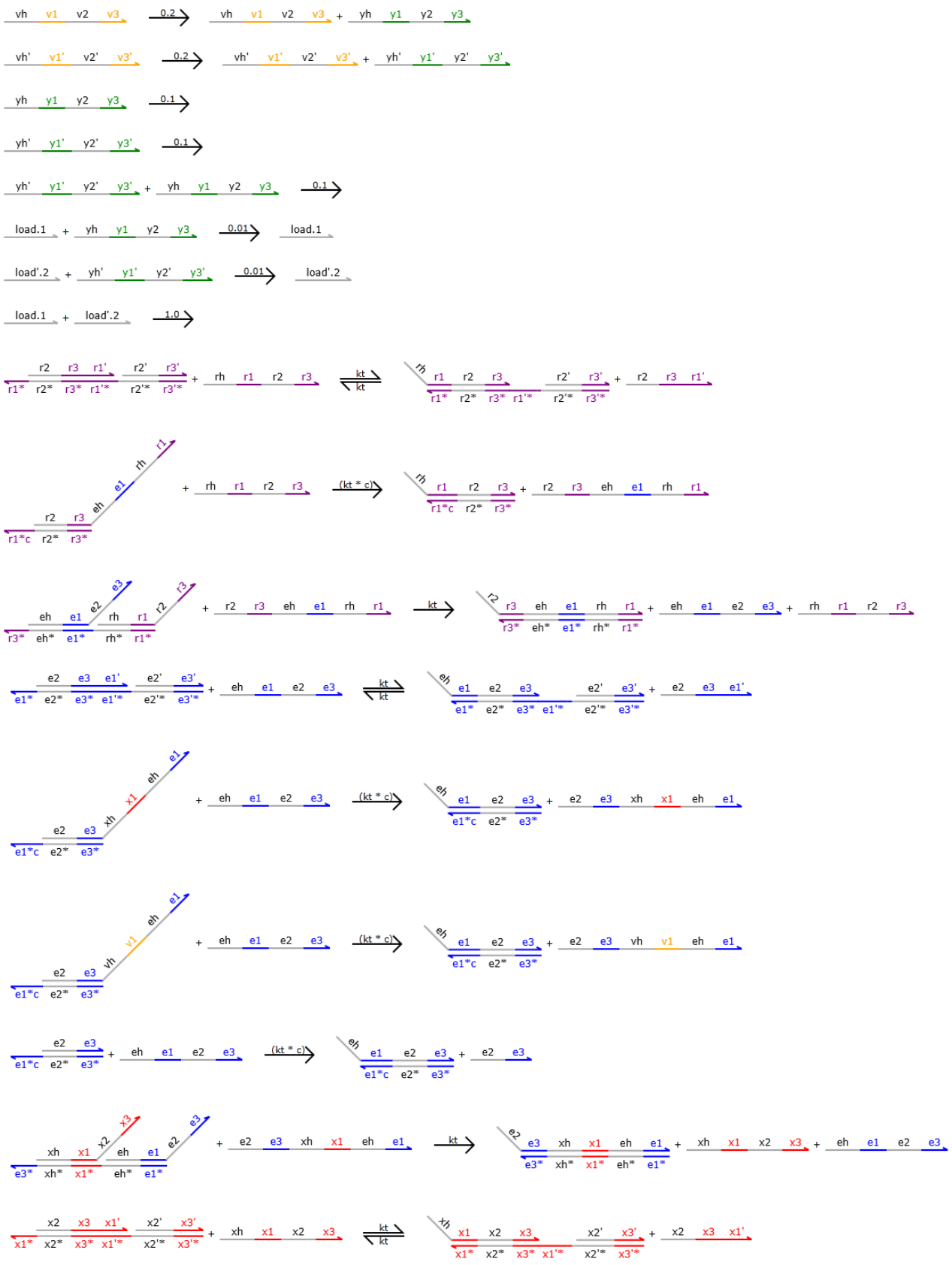


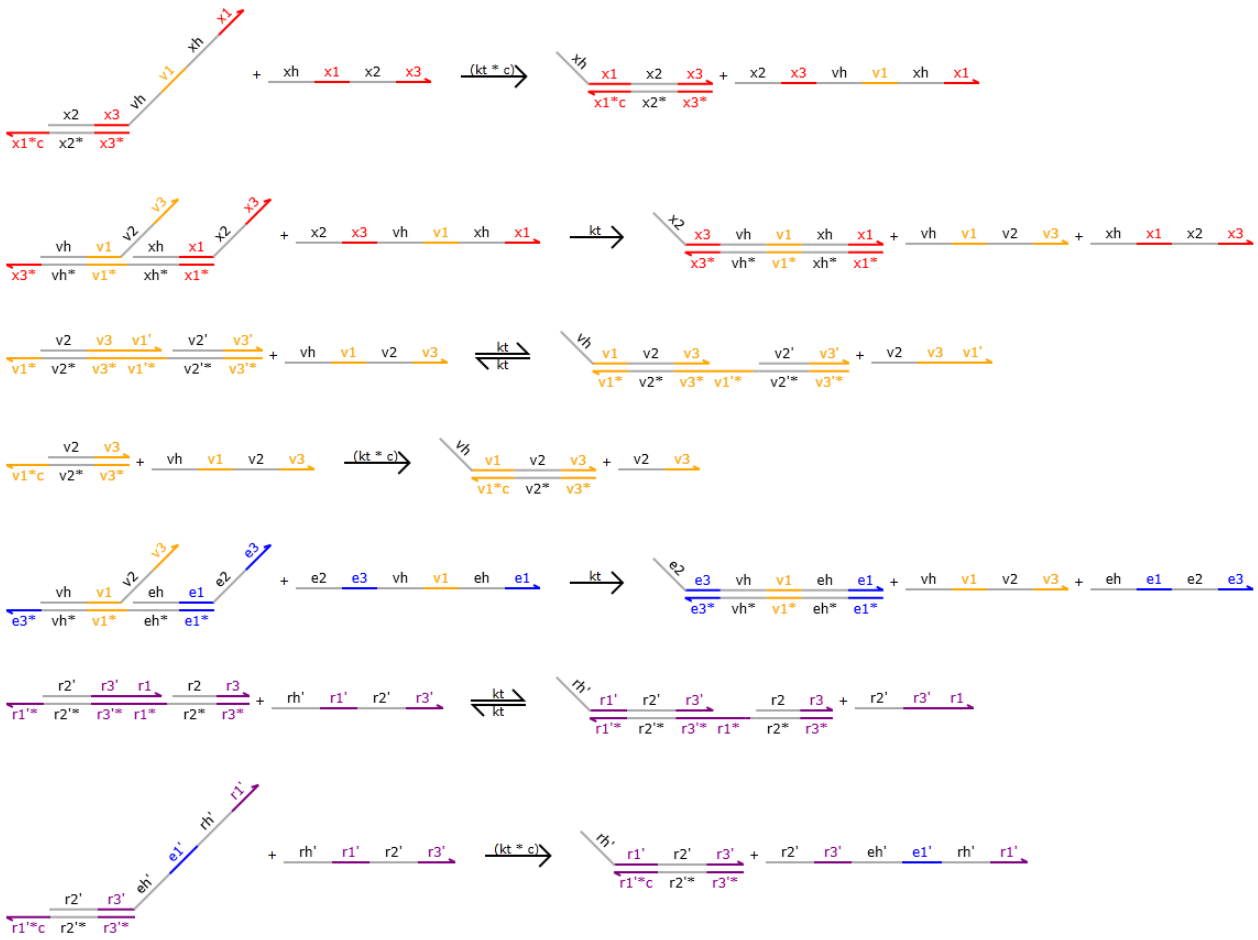
Figure S6: Four domain strand displacement implementation of a PI controller, using the scheme proposed in [8].

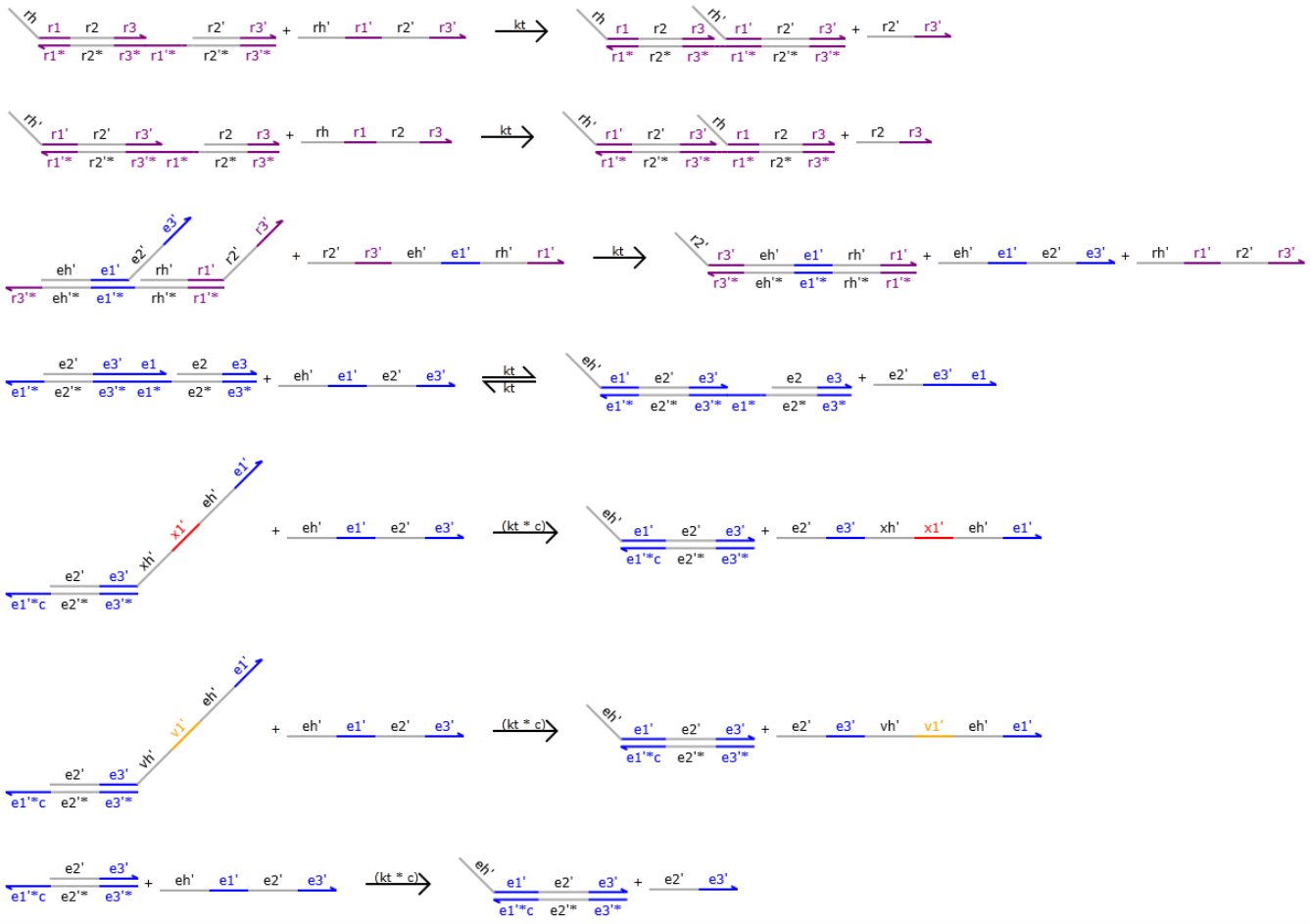
6.2.1 Initial Conditions

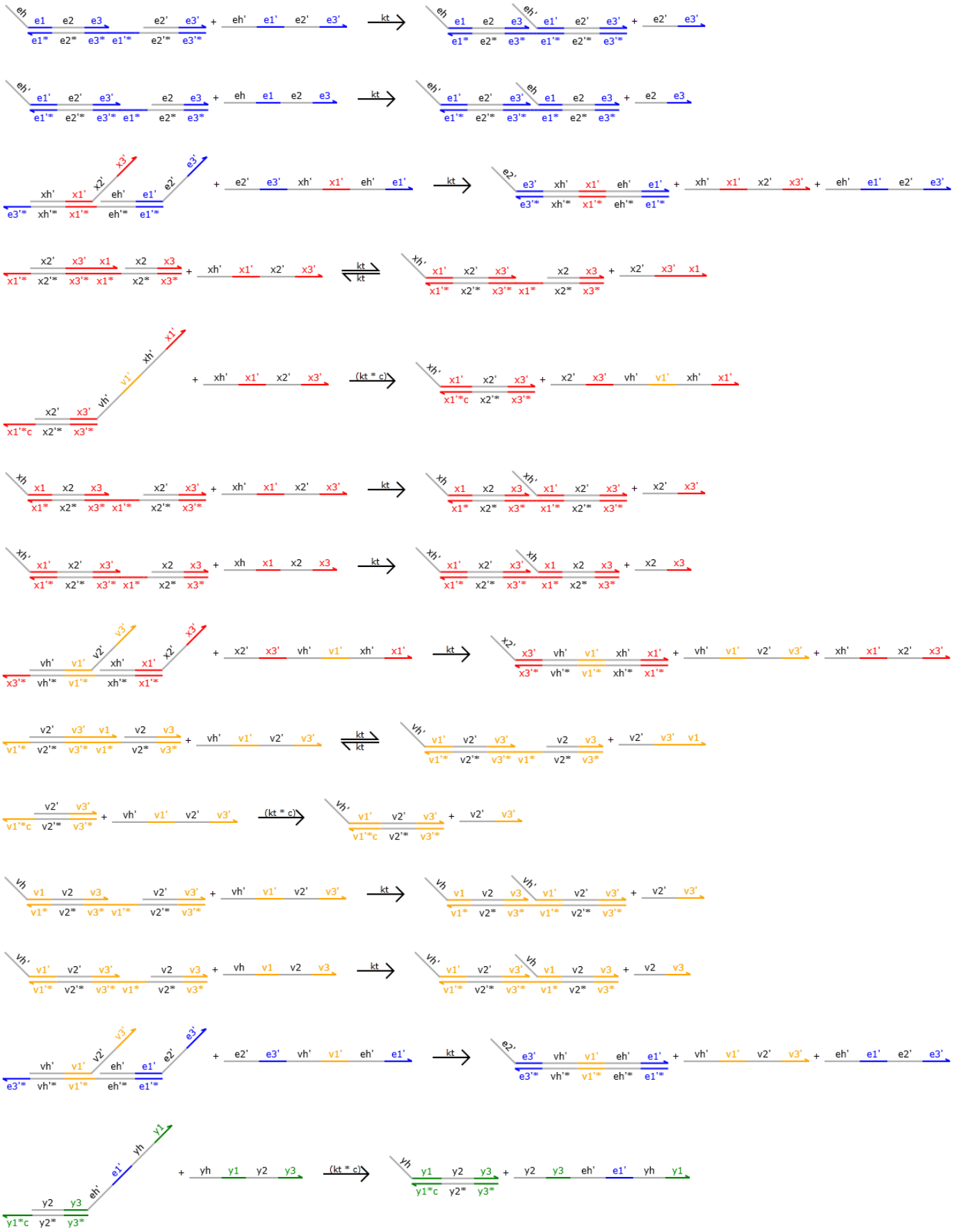


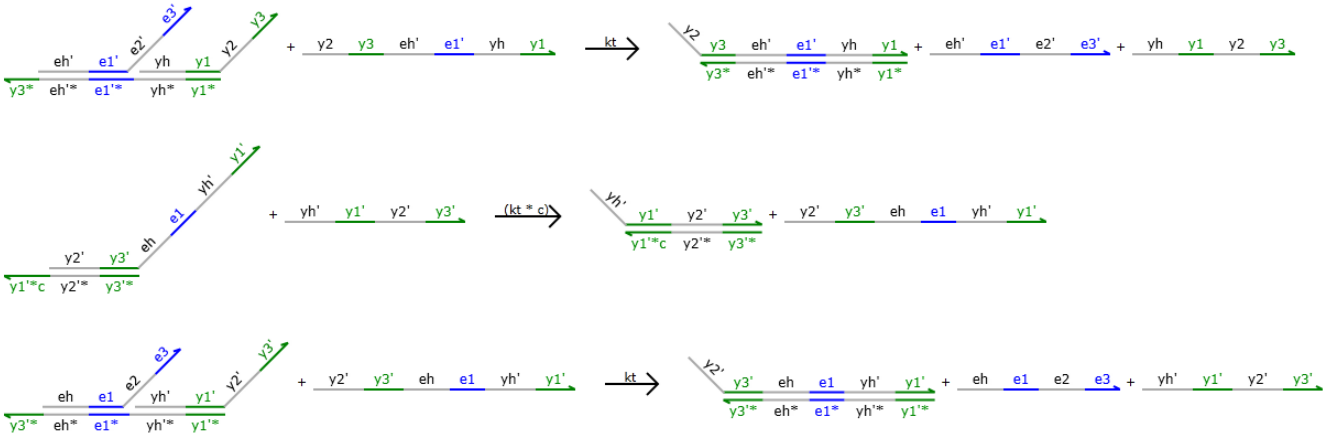
6.2.2 Reactions




x1 xh v1 vh x2 x3
x1*c x2* x3*
xh x1 x2 x3
(kt * c)
xh x1 x2 x3
x1*c x2* x3*
x2 x3 vh v1 xh x1
x3 v3 v2 x2 x3 x1
x3* vh* v1* xh* x1*
x2 x3 vh v1 xh x1
kt
x2 x3 vh v1 xh x1
x3* vh* v1* xh* x1*
vh v1 v2 v3
xh x1 x2 x3
v2 v3 v1' v2' v3'
v1* v2* v3* v1'* v2'* v3'*
vh v1 v2 v3
kt
kt
vh v1 v2 v3 v2' v3'
v1* v2* v3* v1'* v2'* v3'*
v2 v3 v1'
v2 v3
v1*c v2* v3*
vh v1 v2 v3
(kt * c)
vh v1 v2 v3
v1*c v2* v3*
v2 v3
v3 v2 e2 e3
vh v1 eh e1
e3* vh* v1* eh* e1*
e2 e3 vh v1 eh e1
kt
e2 e3 vh v1 eh e1
e3* vh* v1* eh* e1*
vh v1 v2 v3
eh e1 e2 e3
r2' r3' r1 r2 r3
r1'* r2'* r3'* r1* r2* r3*
rh' r1' r2' r3'
kt
kt
rh' r1' r2' r3' r2 r3
r1'* r2'* r3'* r1* r2* r3*
r2' r3' r1
r1' rh' e1' eh'
r2' r3'
r1'*c r2'* r3'*
rh' r1' r2' r3'
(kt * c)
rh' r1' r2' r3'
r1'*c r2'* r3'*
r2' r3' eh' e1' rh' r1'


rh r1 r2 r3 r2' r3'
r1'* r2'* r3'* r1'* r2'* r3'*
rh' r1' r2' r3'
kt
rh' r1' r2' r3'
kt
eh' e1' e2' e3' rh' r1' r2' r3'
kt
e2' e3' e1 e2 e3
eh' e1' e2' e3'
kt
kt
eh' e1' e2' e3'
(kt * c)
eh' e1' e2' e3'
(kt * c)
e2' e3'
eh' e1' e2' e3'
(kt * c)



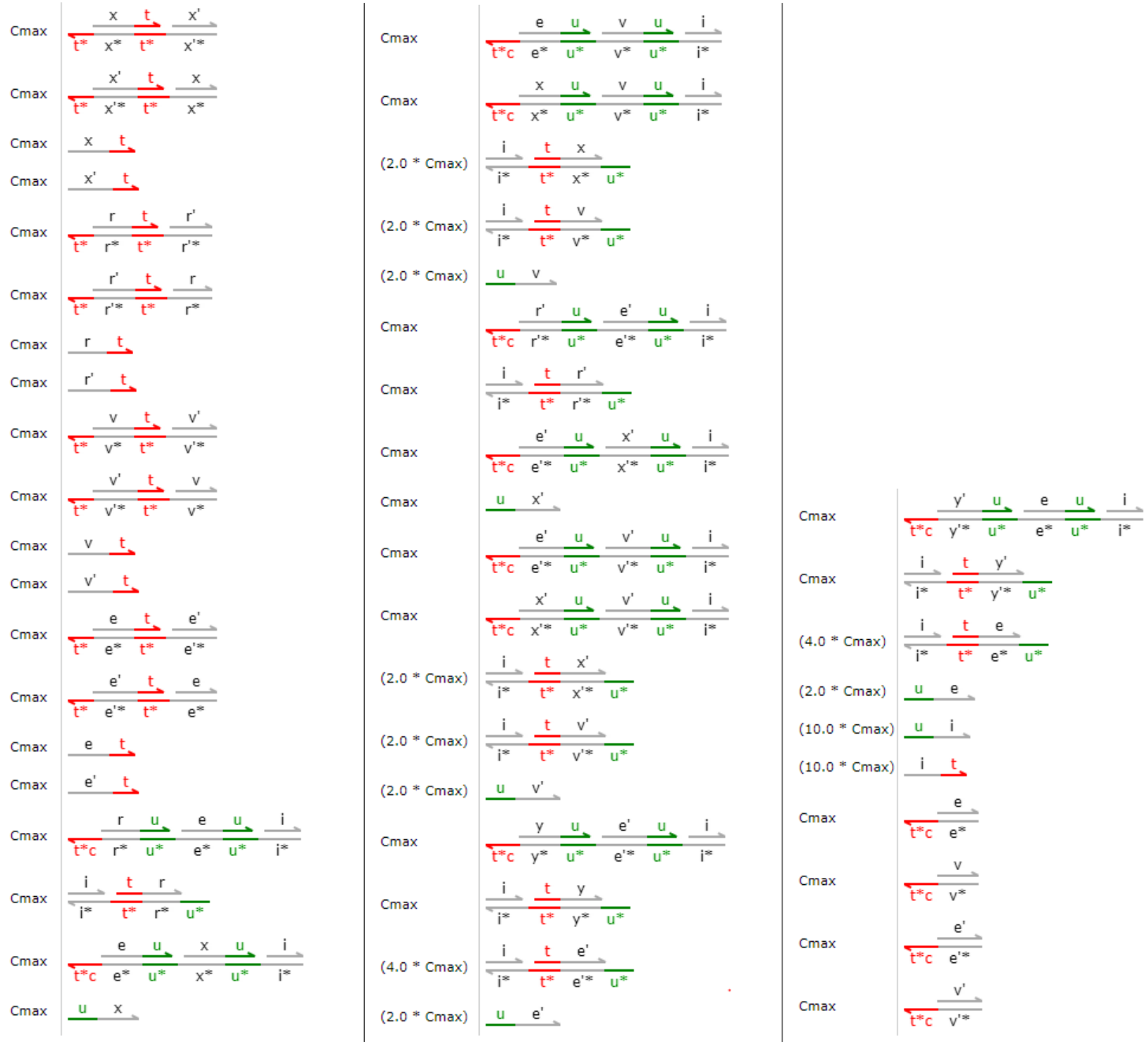

eh' e1' e2' e3'
yh y1 y2 y3
y3* eh'* e1'* yh* y1*
+ y2 y3 eh' e1' yh y1
kt
y2 y3 eh' e1' yh y1
y3* eh'* e1'* yh* y1*
+ eh' e1' e2' e3'
+ yh y1 y2 y3
yh' y1' e1 eh y2' y3'
y1'*c y2'* y3'*
+ yh' y1' y2' y3'
(kt * c)
yh' y1' y2' y3'
y1'*c y2'* y3'*
+ y2' y3' eh e1 yh' y1'
eh e1 e2' e3
yh' y1' y2' y3'
eh* e1* yh'* y1'*
y3'*
+ y2' y3' eh e1 yh' y1'
kt
y2' y3' eh e1 yh' y1'
y3'* eh* e1* yh'* y1'*
+ eh e1 e2 e3
+ yh' y1' y2' y3'

6.3 2Domain DNA Strand Displacement implementation

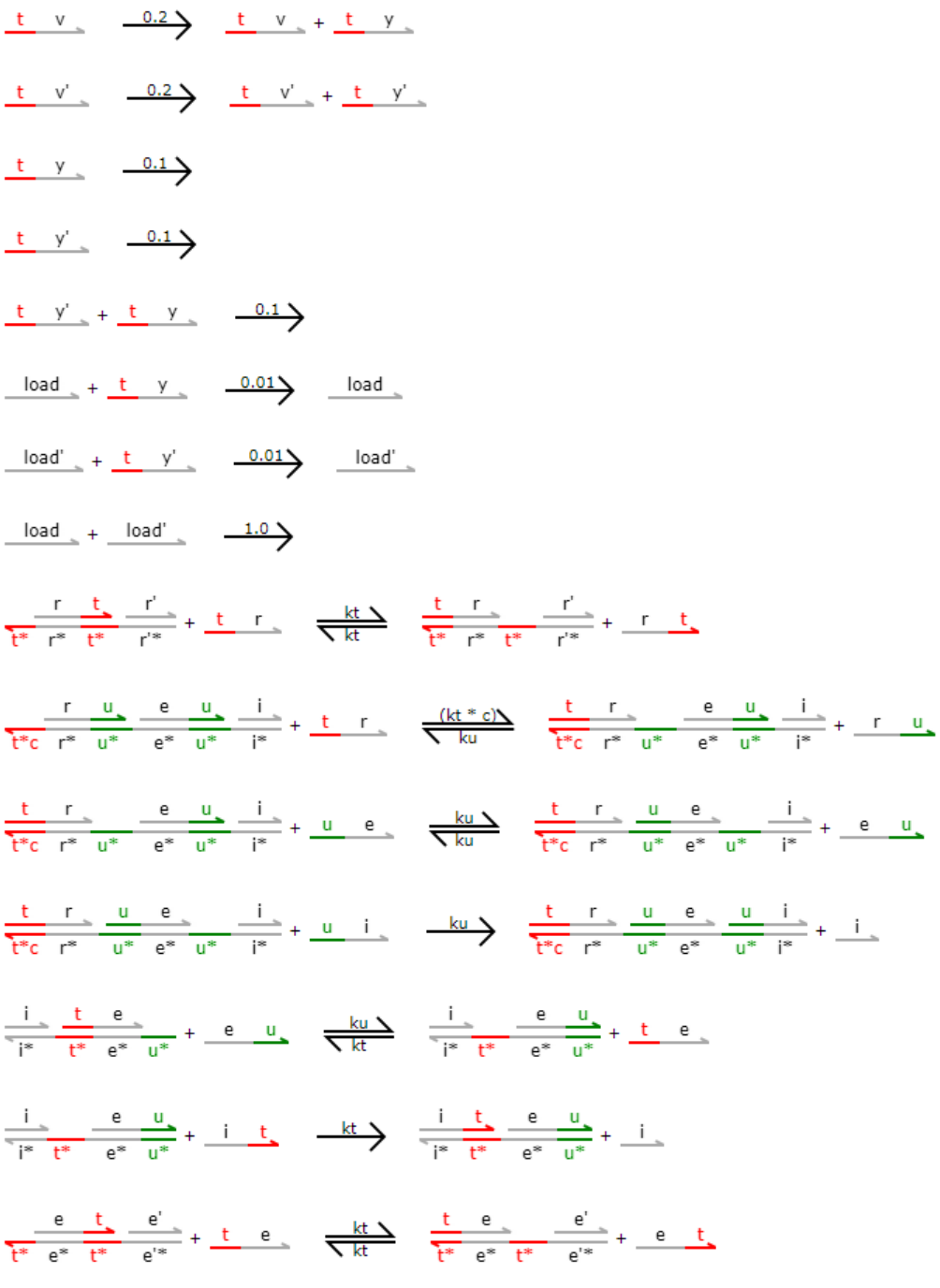
The Visual DSD code for this system is identical to that of the CRN code, but with the CRN section replaced with the following, where all kinetic parameters for this system were obtained from [8] and were consistent with published experimental results [12, 1]

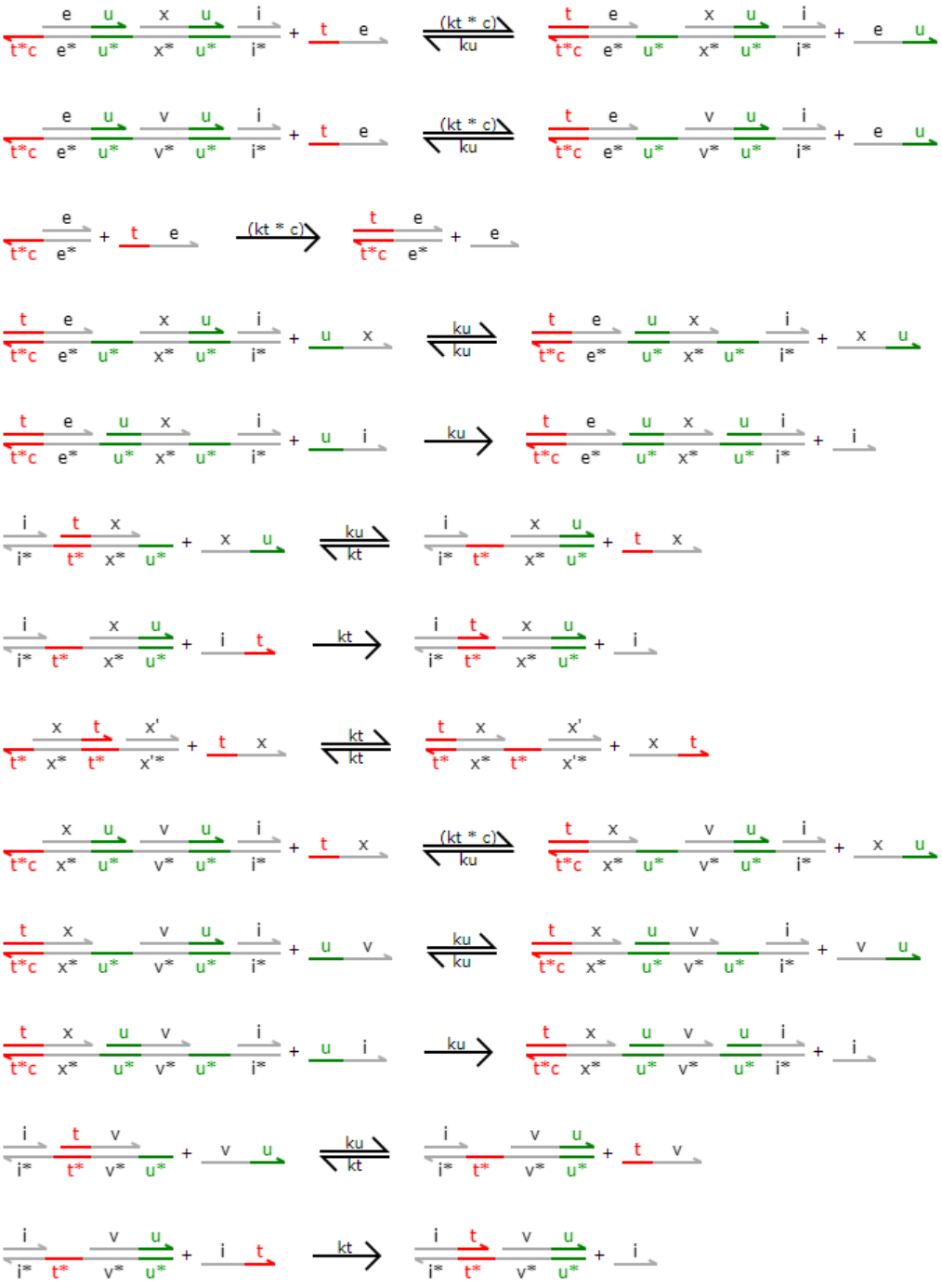
```
directive compilation infinite
new t@kt,unbind
new u@ku,unbind
new x new x' new v new v' new y new y' new r new r'
new e new e' new load new load' new i

def Signal((x,x')) = <t^ x>
def Degradation(k,(x,x')) = Cmax/k * {t^(c)*}[x]
def Annihilation((x,x')) =
( Cmax * {t^*}[x t^]:[x']
| Cmax * {t^*}[x' t^]:[x]
| Cmax * <x t^>
| Cmax * <x' t^>
)
def Catalysis(k,(x,x'),(y,y')) =
( Cmax*k * {t^(c)*}[x u^]:[y u^]:[i]
| Cmax*k * [i]:[t^ x]{u^*}
| Cmax*k * [i]:[t^ y]{u^*}
| Cmax*k * <u^ y>
| Cmax*k * <u^ i>
| Cmax*k * <i t^>
)
def Signal'((x,x')) = Signal((x',x))
def Degradation'(k,(x,x')) = Degradation(k,(x',x))
def Catalysis'(k,(x,x'),(y,y')) = Catalysis(k,(x',x),(y',y))
def CatalysisInv(k,x,(y,y')) = Catalysis(k,x,(y',y))
def CatalysisInv'(k,(x,x'),y) = Catalysis(k,(x',x),y)

def x = (x,x')
def y = (y,y')
def v = (v,v')
def e = (e,e')
def r = (r,r')
```

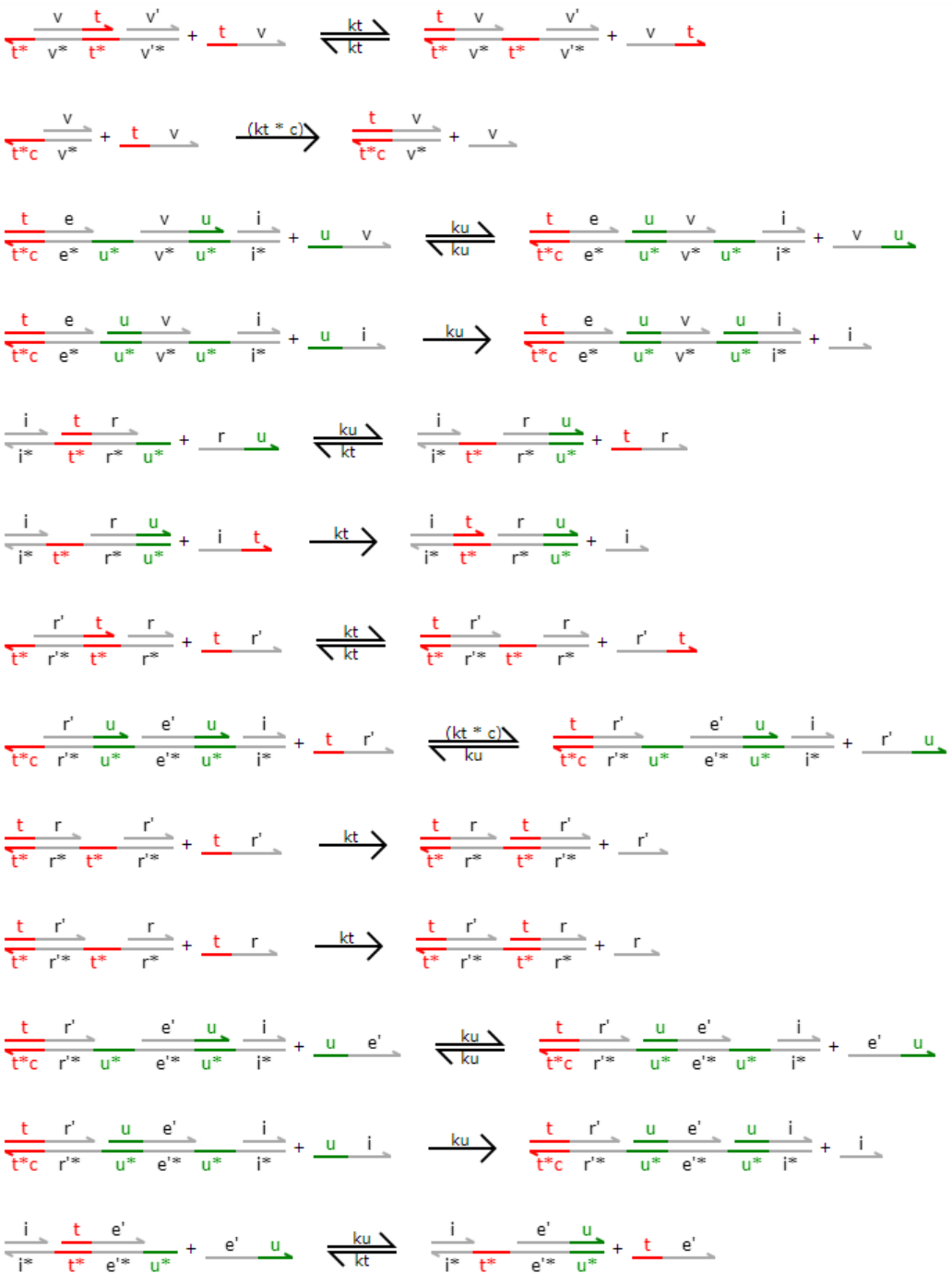
6.3.1 Initial Conditions



6.3.2 Reactions




e u x u i
t*c e* u* x* u* i*
+ t e
(kt * c)
ku
t e x u i
t*c e* u* x* u* i*
+ e u
e u v u i
t*c e* u* v* u* i*
+ t e
(kt * c)
ku
t e v u i
t*c e* u* v* u* i*
+ e u
e
t*c e*
+ t e
(kt * c)
t e
t*c e*
+ e
t e x u i
t*c e* u* x* u* i*
+ u x
ku
ku
t e u x i
t*c e* u* x* u* i*
+ x u
t e u x i
t*c e* u* x* u* i*
+ u i
ku
t e u x u i
t*c e* u* x* u* i*
+ i
i t x
i* t* x* u*
+ x u
ku
kt
i x u
i* t* x* u*
+ t x
i x u
i* t* x* u*
+ i t
kt
i t x u
i* t* x* u*
+ i
x t x'
t* x* t* x'*
+ t x
kt
kt
t x x'
t* x* t* x'*
+ x t
x u v u i
t*c x* u* v* u* i*
+ t x
(kt * c)
ku
t x v u i
t*c x* u* v* u* i*
+ x u
t x v u i
t*c x* u* v* u* i*
+ u v
ku
ku
t x u v i
t*c x* u* v* u* i*
+ v u
t x u v i
t*c x* u* v* u* i*
+ u i
ku
t x u v u i
t*c x* u* v* u* i*
+ i
i t v
i* t* v* u*
+ v u
ku
kt
i v u
i* t* v* u*
+ t v
i v u
i* t* v* u*
+ i t
kt
i t v u
i* t* v* u*
+ i


kt
kt
(kt * c)
ku
ku
ku
kt
kt
kt
(kt * c)
ku
kt
kt
kt
ku
ku
ku
kt


i e' u
i* t* e'* u*
+ i t
kt
i t e' u
i* t* e'* u*
+ i
e' t e
t* e'* t* e*
+ t e'
kt
kt
t e' e
t* e'* t* e*
+ e' t
e' u x' u i
t*c e'* u* x'* u* i*
+ t e'
(kt * c)
ku
t e' x' u i
t*c e'* u* x'* u* i*
+ e' u
e' u v' u i
t*c e'* u* v'* u* i*
+ t e'
(kt * c)
ku
t e' v' u i
t*c e'* u* v'* u* i*
+ e' u
e'
t*c e'*
+ t e'
(kt * c)
t e'
t*c e'*
+ e'
t e e' t e'
t* e* t* e'*
+ t e'
kt
t e t e'
t* e* t* e'*
+ e'
t e' e t e
t* e'* t* e*
+ t e
kt
t e' t e
t* e'* t* e*
+ e
t e' x' u i u x'
t*c e'* u* x'* u* i*
ku
ku
t e' u x' i
t*c e'* u* x'* u* i*
+ x' u
t e' u x' i u i
t*c e'* u* x'* u* i*
ku
t e' u x' u i
t*c e'* u* x'* u* i*
+ i
i t x' x' u
i* t* x'* u*
+ x' u
ku
kt
i x' u
i* t* x'* u*
+ t x'
i x' u i t
i* t* x'* u*
+ i
kt
i t x' u
i* t* x'* u*
+ i
x' t x x' t
t* x'* t* x*
+ t x'
kt
kt
t x' x
t* x'* t* x*
+ x' t
x' u v' u i t x'
t*c x'* u* v'* u* i*
(kt * c)
ku
t x' v' u i
t*c x'* u* v'* u* i*
+ x' u


kt
ku
ku
kt
kt
kt
(kt * c)
kt
kt
ku
ku
kt


i r' u
i* t* r'* u*
+ i t
kt
i t r' u
i* t* r'* u*
+ i
y u e' u i
t*c y* u* e'* u* i*
+ t y
(kt * c)
ku
t y e' u i
t*c y* u* e'* u* i*
+ y u
t y e' u i
t*c y* u* e'* u* i*
+ u e'
ku
ku
t y u e' i
t*c y* u* e'* u* i*
+ e' u
t y u e' i
t*c y* u* e'* u* i*
+ u i
ku
t y u e' u i
t*c y* u* e'* u* i*
+ i
i t y
i* t* y* u*
+ y u
ku
kt
i y u
i* t* y* u*
+ t y
i y u
i* t* y* u*
+ i t
kt
i t y u
i* t* y* u*
+ i
y' u e u i
t*c y'* u* e* u* i*
+ t y'
(kt * c)
ku
t y' e u i
t*c y'* u* e* u* i*
+ y' u
t y' e u i
t*c y'* u* e* u* i*
+ u e
ku
ku
t y' u e i
t*c y'* u* e* u* i*
+ e u
t y' u e u i
t*c y'* u* e* u* i*
+ u i
ku
t y' u e u i
t*c y'* u* e* u* i*
+ i
i t y'
i* t* y'* u*
+ y' u
ku
kt
i y' u
i* t* y'* u*
+ t y'
i y' u
i* t* y'* u*
+ i t
kt
i t y' u
i* t* y'* u*
+ i

6.4 DNA Enzyme implementation

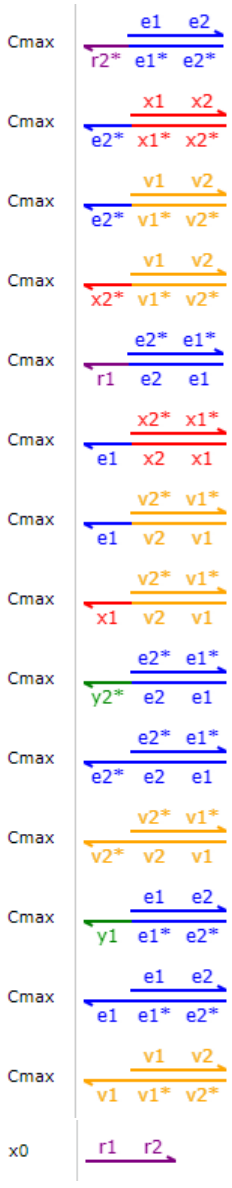
6.4.1 Visual DSD Code

The Visual DSD code for this system is identical to that of the CRN code, but with the CRN section replaced with the following:

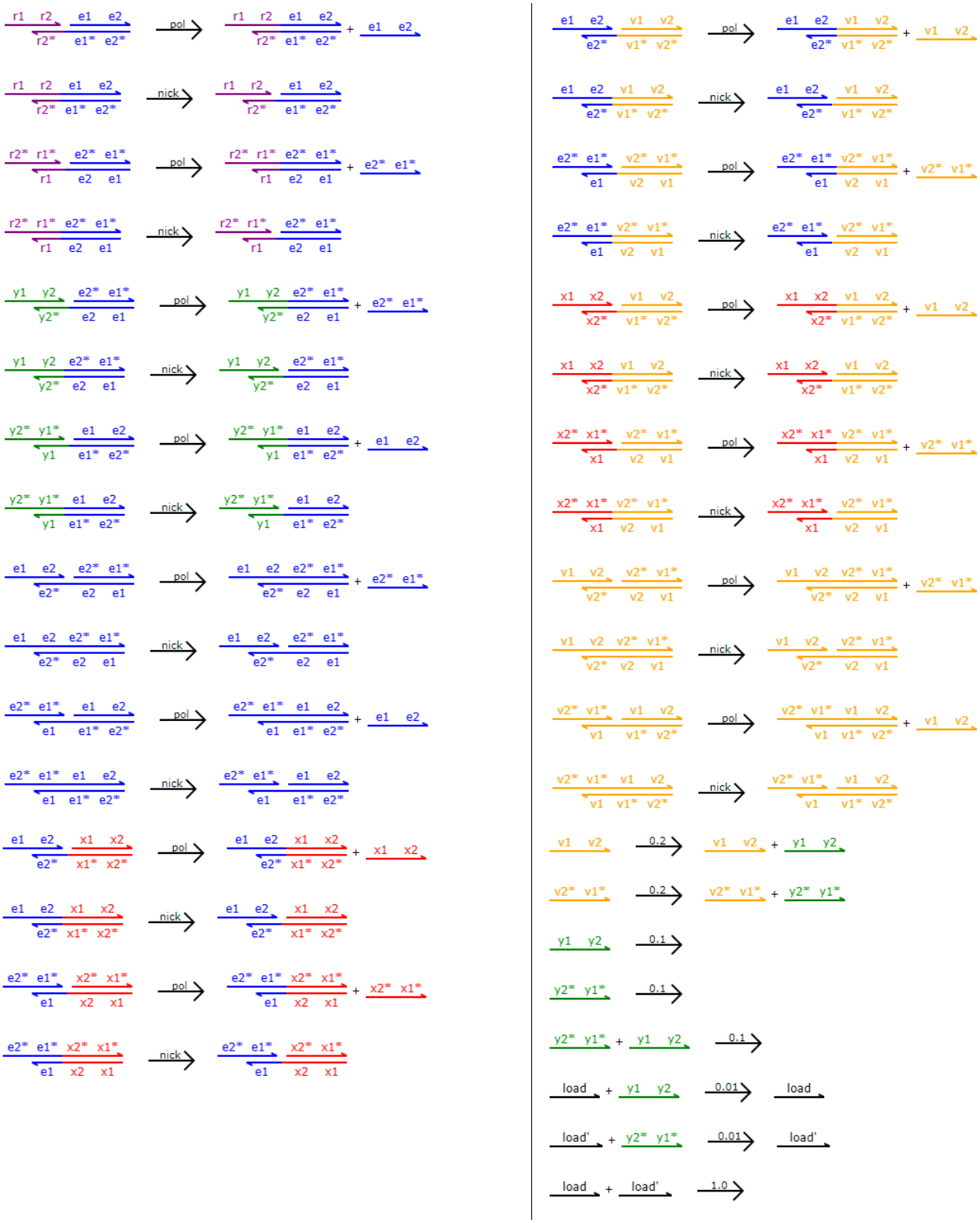
```
directive compilation default
directive unproductive

dom x1 = {colour = "red"; bind = bind; unbind = unbind}
dom x2 = {colour = "red"; bind = bind; unbind = unbind}
dom r1 = {colour = "purple"; bind = bind; unbind = unbind}
dom r2 = {colour = "purple"; bind = bind; unbind = unbind}
dom e1 = {colour = "blue"; bind = bind; unbind = unbind}
dom e2 = {colour = "blue"; bind = bind; unbind = unbind}
dom v1 = {colour = "orange"; bind = bind; unbind = unbind}
dom v2 = {colour = "orange"; bind = bind; unbind = unbind}
dom y1 = {colour = "green"; bind = bind; unbind = unbind}
dom y2 = {colour = "green"; bind = bind; unbind = unbind}
dom load = {colour = "black"}
dom load' = {colour = "black"}
dom dx = {bind = ann; subdomains = [x1;x2]}
dom dr = {bind = ann; subdomains = [r1;r2]}
dom de = {bind = ann; subdomains = [e1;e2]}
dom dv = {bind = ann; subdomains = [v1;v2]}
dom dy = {bind = ann; subdomains = [y1;y2]}

def Signal((x1,x2)) = <x1^ x2^>
def Signal'((x1,x2)) = <x2^* x1^*>
def Template((x1,x2),(y1,y2)) = <x1^>[x2^]:[y1^ y2^]
def Template'((x1,x2),(y1,y2)) = <x2^*>[x1^*]:[y2^* y1^*]
def TemplateOff((x1,x2),(y1,y2)) = {x2^*}[y1^ y2^]
def TemplateOff'((x1,x2),(y1,y2)) = {x1^}[y2^* y1^*]
def TemplateDS((x1,x2),(y1,y2)) = <x1^>[x2^ y1^ y2^]
def TemplateDS'((x1,x2),(y1,y2)) = <x2^*>[x1^* y2^* y1^*]
def TemplateOffInv((x1,x2),(y1,y2)) = {x2^*}[y2^* y1^*]
def TemplateOffInv'((x1,x2),(y1,y2)) = {x1^}[y1^ y2^]
def TemplateInv((x1,x2),(y1,y2)) = <x1^>[x2^]:[y2^* y1^*]
def TemplateInv'((x1,x2),(y1,y2)) = <x2^*>[x1^*]:[y1^ y2^]
def TemplateDSInv((x1,x2),(y1,y2)) = <x1^>[x2^ y2^* y1^*]
def TemplateDSInv'((x1,x2),(y1,y2)) = <x2^*>[x1^* y1^ y2^]
def Ann((x1,x2)) = [x1^ x2^]

def r = (r1,r2)
def e = (e1,e2)
def v = (v1,v2)
def y = (y1,y2)
def x = (x1,x2)

def Catalysis(k,x,y) =
( Cmax*k * TemplateOff(x,y)
| rxn Template(x,y) ->{pol} TemplateDS(x,y) + Signal(y)
| rxn TemplateDS(x,y) ->{nick} Template(x,y)
| 0 * Template(x,y) | 0 * TemplateDS(x,y)
)
def Catalysis'(k,x,y) =
( Cmax*k * TemplateOff'(x,y)
| rxn Template'(x,y) ->{pol} TemplateDS'(x,y) + Signal'(y)
| rxn TemplateDS'(x,y) ->{nick} Template'(x,y)
)
def CatalysisInv(k,x,y) =
( Cmax*k * TemplateOffInv(x,y)
```

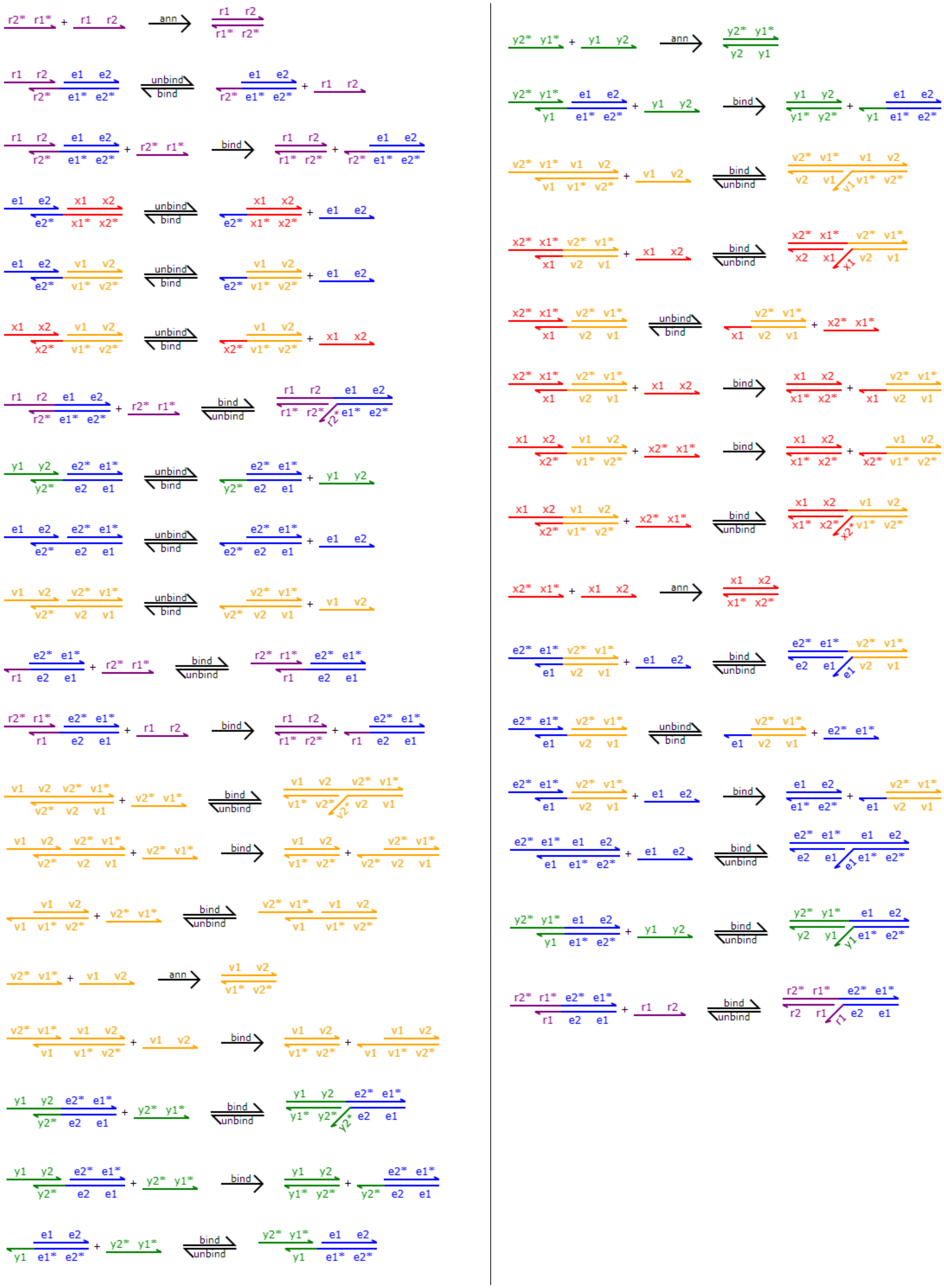
```
| rxn TemplateInv(x,y) ->{pol} TemplateDSInv(x,y) + Signal'(y)
| rxn TemplateDSInv(x,y) ->{nick} TemplateInv(x,y)
| 0 * TemplateInv(x,y) | 0 * TemplateDSInv(x,y)
)
def CatalysisInv'(k,x,y) =
( Cmax*k * TemplateOffInv'(x,y)
| rxn TemplateInv'(x,y) ->{pol} TemplateDSInv'(x,y) + Signal(y)
| rxn TemplateDSInv'(x,y) ->{nick} TemplateInv'(x,y)
)
def Annihilation(x) =  0*Ann(x)
def Degradation(k,x) = CatalysisInv(1.0/k,x,x)
def Degradation'(k,x) = CatalysisInv'(1.0/k,x,x)
```

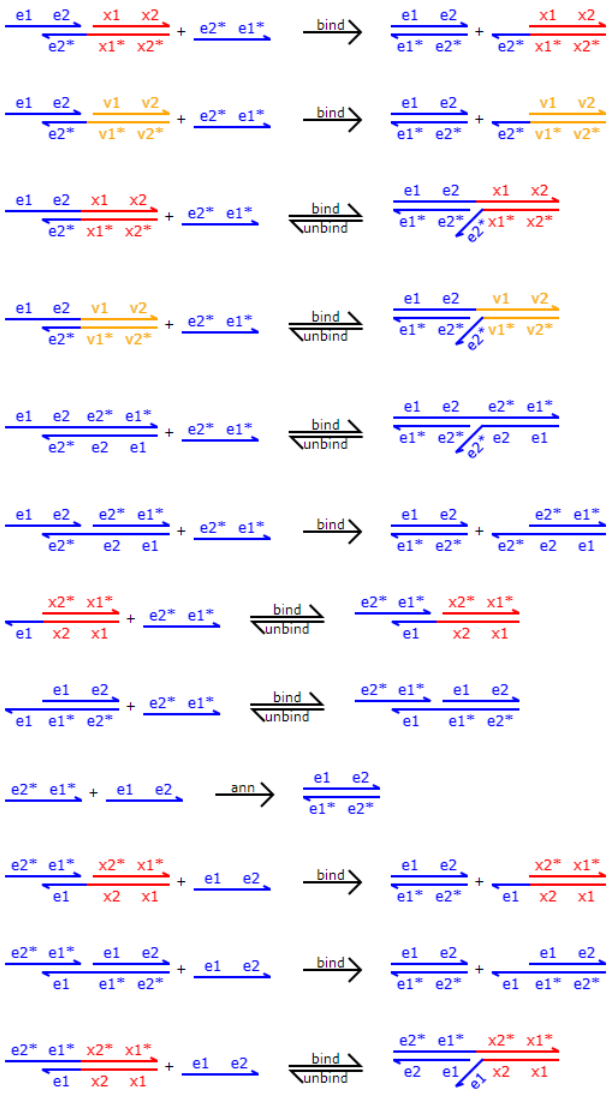
6.4.2 Initial Conditions



6.4.3 Reactions






bind
bind
unbind
bind
unbind
bind
unbind
bind
unbind
ann
bind
bind
bind
unbind

6.5 RNA Enzyme implementation

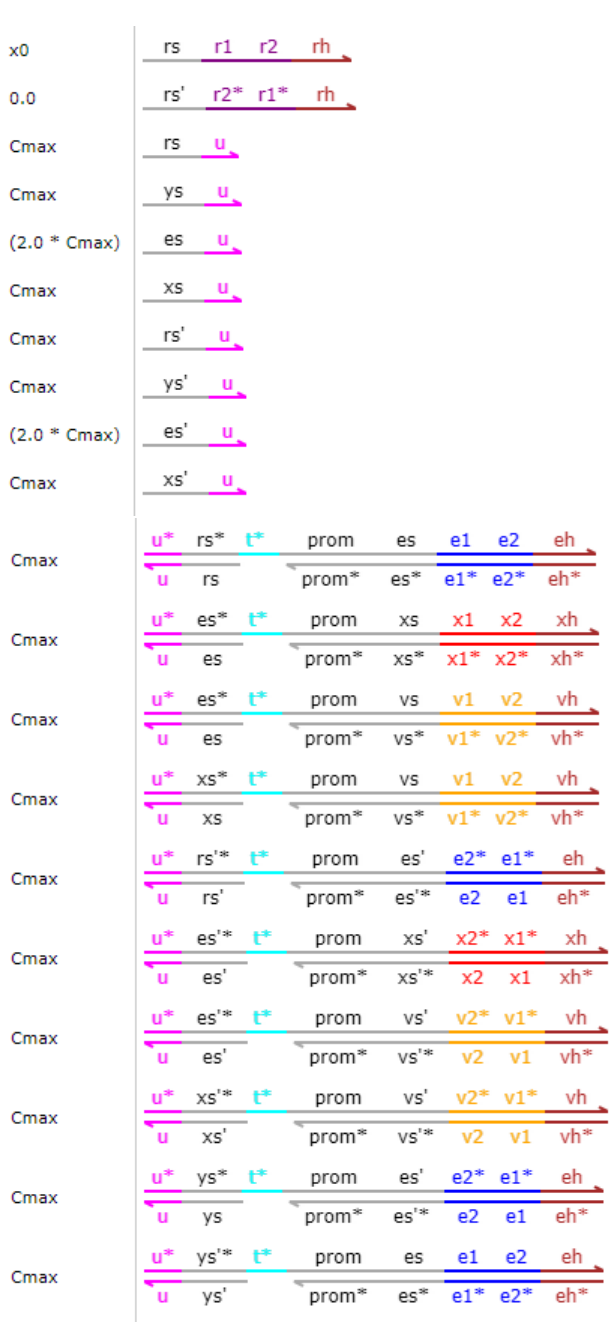
6.5.1 Visual DSD Code

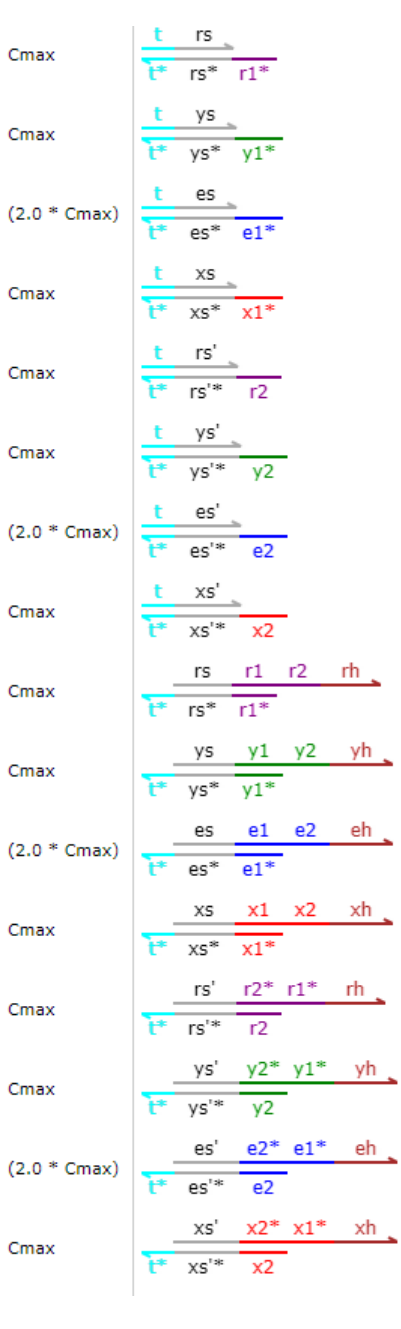
The Visual DSD code for this system is identical to that of the CRN code, but with the CRN section replaced with the following:

```
directive compilation infinite
def kbind = bind1
dom x1 = {colour = "red"; bind = kbind; unbind = unbind}
dom x2 = {colour = "red"; bind = kbind; unbind = unbind}
dom r1 = {colour = "purple"; bind = kbind; unbind = unbind}
dom r2 = {colour = "purple"; bind = kbind; unbind = unbind}
dom e1 = {colour = "blue"; bind = kbind; unbind = unbind}
dom e2 = {colour = "blue"; bind = kbind; unbind = unbind}
dom v1 = {colour = "orange"; bind = kbind; unbind = unbind}
dom v2 = {colour = "orange"; bind = kbind; unbind = unbind}
dom y1 = {colour = "green"; bind = kbind; unbind = unbind}
dom y2 = {colour = "green"; bind = kbind; unbind = unbind}
dom t = {colour = "cyan"; bind = bind2; unbind = unbind}
dom u = {colour = "magenta"; bind = bind2; unbind = unbind}
dom rh = {colour = "brown"}
dom eh = {colour = "brown"}
dom vh = {colour = "brown"}
dom xh = {colour = "brown"}
dom yh = {colour = "brown"}

new rs new rs' new es new es' new vs new vs'
new xs new xs' new ys new ys' new load new load' new prom

def Signal((x1,x2,xs,xs',xh)) = <x1^ x2^ xs xh>
def Signal'((x1,x2,xs,xs',xh)) = <x2^* x1^* xs' xh>
def Template((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = {u^*}[xs t^]:[prom y1^ y2^ ys yh]
def Template'((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = {u^*}[xs' t^]:[prom y2^* y1^* ys' yh]
def TemplateOff((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = [u^ xs]{t^*}:[prom y1^ y2^ ys yh]
def TemplateOff'((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = [u^ xs']{t^*}:[prom y2^* y1^* ys' yh]
def TemplateOffInv((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = [u^ xs]{t^*}:[prom y2^* y1^* ys' yh]
def TemplateOffInv'((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = [u^ xs']{t^*}:[prom y1^ y2^ ys yh]
def TemplateInv((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = {u^*}[xs t^]:[prom y2^* y1^* ys' yh]
def TemplateInv'((x1,x2,xs,xs',xh),(y1,y2,ys,ys',yh)) = {u^*}[xs' t^]:[prom y1^ y2^ ys yh]
def Transducer((x1,x2,xs,xs',xh)) = {x2^*}[xs t^]
def Transducer'((x1,x2,xs,xs',xh)) = {x1^}[xs' t^]
def TransducerRev((x1,x2,xs,xs',xh)) = <x1^>[x2^ xs]<xh>{t^*}
def TransducerRev'((x1,x2,xs,xs',xh)) = <x2^*>[x1^* xs']<xh>{t^*}
def Rev((x1,x2,xs,xs',xh)) = <u^ xs>
def Rev'((x1,x2,xs,xs',xh)) = <u^ xs'>
def Fwd(t,(x1,x2,xs,xs',xh)) = <xs t^>
def Fwd'(t,(x1,x2,xs,xs',xh)) = <xs' t^>
def Ann((x1,x2,xs,xs',xh)) = {xh xs'}[x1^ x2^]<xs xh>

def r = (r1,r2,rs,rs',rh)
def e = (e1,e2,es,es',eh)
def v = (v1,v2,vs,vs',vh)
def y = (y1,y2,ys,ys',yh)
def x = (x1,x2,xs,xs',xh)

def Catalysis(k,x,y) =
( Cmax*k * TemplateOff(x,y)
| Cmax*k * Transducer(x)
| Cmax*k * TransducerRev(x)
| Cmax*k * Rev(x)
| rxn Template(x,y) ->{pol} Template(x,y) + Signal(y)
| 0 * Template(x,y) | 0* Fwd(t,x)
```

```
)
def Catalysis'(k,x,y) =
( Cmax*k * TemplateOff'(x,y)
| Cmax*k * Transducer'(x)
| Cmax*k * TransducerRev'(x)
| Cmax*k * Rev'(x)
| rxn Template'(x,y) ->{pol} Template'(x,y) + Signal'(y)
)
def CatalysisInv(k,x,y) =
( Cmax*k * TemplateOffInv(x,y)
| Cmax*k * Transducer(x)
| Cmax*k * TransducerRev(x)
| Cmax*k * Rev(x)
| rxn TemplateInv(x,y) ->{pol} TemplateInv(x,y) + Signal'(y)
)
def CatalysisInv'(k,x,y) =
( Cmax*k * TemplateOffInv'(x,y)
| Cmax*k * Transducer'(x)
| Cmax*k * TransducerRev'(x)
| Cmax*k * Rev'(x)
| rxn TemplateInv'(x,y) ->{pol} TemplateInv'(x,y) + Signal(y)
)
def Degradation(k,x) = rxn Signal(x) ->{deg/k}
def Degradation'(k,x) = rxn Signal'(x) ->{deg/k}
def Annihilation(x) = 0*Ann(x)
```

6.5.2 Initial Conditions





6.5.3 Reactions


rs' r2* r1* rh + rs r1 r2 rh
ann
rs r1 r2 rh
rh r1* r2* rs'
u* rs* t* prom es e1 e2 eh + rs u
rs t prom* es* e1* e2* eh*
bind2 bind2
u* rs* t* prom es e1 e2 eh + t rs
u rs prom* es* e1* e2* eh*
u* es* t* prom xs x1 x2 xh + es u
es t prom* xs* x1* x2* xh*
bind2 bind2
u* es* t* prom xs x1 x2 xh + t es
u es prom* xs* x1* x2* xh*
u* es* t* prom vs v1 v2 vh + es u
es t prom* vs* v1* v2* vh*
bind2 bind2
u* es* t* prom vs v1 v2 vh + t es
u es prom* vs* v1* v2* vh*
u* xs* t* prom vs v1 v2 vh + xs u
xs t prom* vs* v1* v2* vh*
bind2 bind2
u* xs* t* prom vs v1 v2 vh + t xs
u xs prom* vs* v1* v2* vh*
t rs + rs r1 r2 rh
t* rs* r1*
bind1 bind2
rs r1 r2 rh + t rs
t* rs* r1*
t rs' + rs' r2* r1* rh
t* rs'* r2
bind1 bind2
rs' r2* r1* rh + t rs'
t* rs'* r2
u* rs'* t* prom es' e2* e1* eh + t rs'
u rs' prom* es'* e2 e1 eh*
bind2 bind2
u* rs'* t* prom es' e2* e1* eh + rs' u
rs' t prom* es'* e2 e1 eh*
es e1 e2 eh + t es
t* es* e1*
bind2 bind1
t es + es e1 e2 eh
t* es* e1*
xs x1 x2 xh + t xs
t* xs* x1*
bind2 bind1
t xs + xs x1 x2 xh
t* xs* x1*
vs' v2* v1* vh + vs v1 v2 vh
ann
vs v1 v2 vh
vh v1* v2* vs'
t ys + ys y1 y2 yh
t* ys* y1*
bind1 bind2
ys y1 y2 yh + t ys
t* ys* y1*


u* ys* t* prom es' e2* e1* eh
u ys prom* es'* e2 e1 eh*
+ t ys
bind2 bind2
u* ys* t* prom es' e2* e1* eh
ys t prom* es'* e2 e1 eh*
+ ys u
t ys'
t* ys'* y2
+ ys' y2* y1* yh
bind1 bind2
ys' y2* y1* yh
t* ys'* y2
+ t ys'
ys' y2* y1* yh + ys y1 y2 yh
ann
ys' y2* y1* yh
yh y2 y1 ys
u* ys'* t* prom es e1 e2 eh
u ys' prom* es* e1* e2* eh*
+ t ys'
bind2 bind2
u* ys'* t* prom es e1 e2 eh
ys' t prom* es* e1* e2* eh*
+ ys' u
u* xs'* t* prom vs' v2* v1* vh
xs' t prom* vs'* v2 v1 vh*
+ xs' u
bind2 bind2
u* xs'* t* prom vs' v2* v1* vh
u xs' prom* vs'* v2 v1 vh*
+ xs' t
xs' x2* x1* xh
t* xs'* x2
+ xs' t
bind2 bind1
t xs'
t* xs'* x2
+ xs' x2* x1* xh
xs' x2* x1* xh + xs x1 x2 xh
ann
xs' x2* x1* xh
xh x1* x2* xs'
u* es'* t* prom vs' v2* v1* vh
es' t prom* vs'* v2 v1 vh*
+ es' u
bind2 bind2
u* es'* t* prom vs' v2* v1* vh
u es' prom* vs'* v2 v1 vh*
+ es' t
u* es'* t* prom xs' x2* x1* xh
u es' prom* xs'* x2 x1 xh*
+ es' t
bind2 bind2
u* es'* t* prom xs' x2* x1* xh
es' t prom* xs'* x2 x1 xh*
+ es' u
es' e2* e1* eh
t* es'* e2
+ es' t
bind2 bind1
t es'
t* es'* e2
+ es' e2* e1* eh
es' e2* e1* eh + es e1 e2 eh
ann
es' e2* e1* eh
eh e1* e2* es'
vs v1 v2 vh
0.2
vs v1 v2 vh + ys y1 y2 yh
vs' v2* v1* vh
0.2
vs' v2* v1* vh + ys' y2* y1* yh
ys y1 y2 yh
0.1
ys' y2* y1* yh
0.1
ys' y2* y1* yh + ys y1 y2 yh
0.1
load + ys y1 y2 yh
0.01
load
load' + ys' y2* y1* yh
0.01
load'
load + load'
1.0

References

[1] Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature nanotechnology*, 8(10):755–762, October 2013.

[2] R Dorf and R Bishop. *Modern Control Systems (12th Edition)*. Prentice Hall, Englewood, N.J., 2011.

[3] Jongmin Kim and Erik Winfree. Synthetic in vitro transcriptional oscillators. *Molecular Systems Biology*, 7:465, Feb 2011.

[4] Matthew R Lakin, Simon Youssef, Luca Cardelli, and Andrew Phillips. Abstractions for DNA circuit design. *Journal of the Royal Society, Interface / the Royal Society*, 9(68):470–86, March 2012.

[5] Matthew R Lakin, Simon Youssef, Filippo Polo, Stephen Emmott, and Andrew Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3, 2011.

[6] N Minorsky. Directional stability of automatically steered bodies. *Journal of the American Society of Naval Engineering*, 34(2), 1922.

[7] Kevin Montagne, Raphael Plasson, Yasuyuki Sakai, Teruo Fujii, and Yannick Rondelez. Programming an in vitro DNA oscillator using a molecular networking strategy. *Molecular systems biology*, 7(466):466, February 2011.

[8] K Oishi and E Klavins. Biomolecular implementation of linear I/O systems. *IET Systems Biology*, 5(4):252–260, 2011.

[9] K J Åström and T Hägglund. *PID Controller: Theory, Design, and Tuning*. ISA – The Instrumentation, Systems, and Automation Society, Research Triangle Park, 2005.

[10] David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proc Natl Acad Sci U S A*, 107(12):5393–5398, Mar 2010.

[11] K. Tan, Q-G. Wang, and C. Hang. *Advances in PID Control*. Springer Verlag, London, U.K., 1999.

[12] D. Y. Zhang and E. Winfree. Control of DNA strand displacement kinetics using toehold exchange. *Journal of the Americal Chemical Society*, 131(47):17303–17314, 2009.